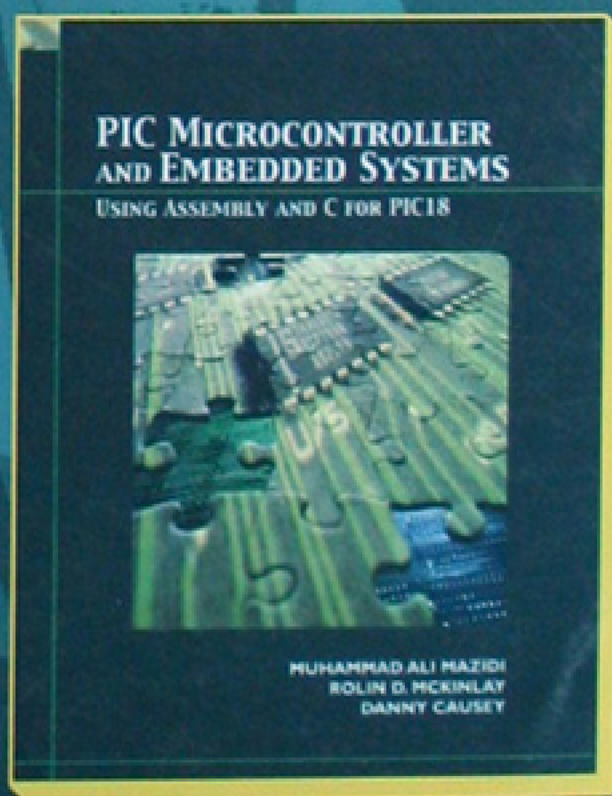
TURING 图灵电子与电气工程丛书

PEARSON
Prentice Hall

PIC技术宝典

PIC Microcontroller and Embedded Systems

Muhammad Ali Mazidi
[美] Rolin D. McKinlay 著
Danny Causey
李中华 张雨浓 陈卓怡 等译



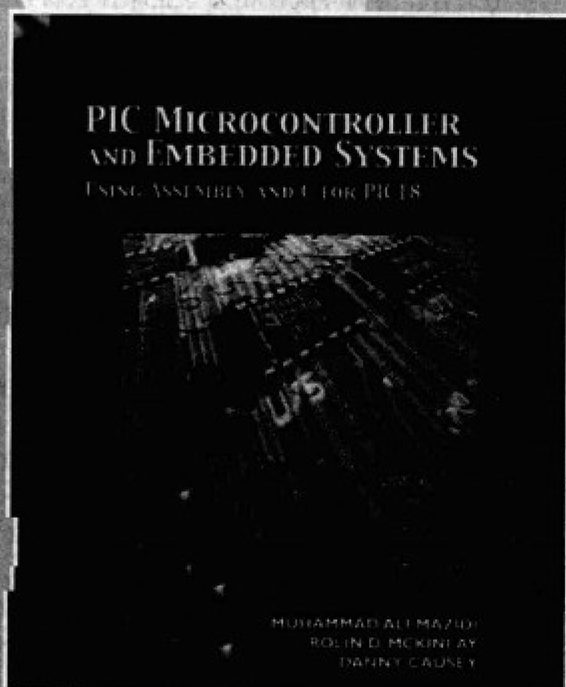
人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵电子与电气工程丛书

PIC技术宝典

PIC Microcontroller and Embedded Systems

Muhammad Ali Mazidi
[美] Rolin D. McKinlay 著
Danny Causey
李中华 张雨浓 陈卓怡 等译



人民邮电出版社
北 京

图书在版编目(CIP)数据

PIC技术宝典/(美)马兹迪(Mazidi,M. A.),(美)麦金莱(McKinlay,R. D.),(美)考西(Causey,D.)著;李中华等译.—北京:人民邮电出版社,2008.10
(图灵电子与电气工程丛书)
书名原文:PIC Microcontroller and Embedded Systems
ISBN 978-7-115-18554-9

Ⅰ.P… Ⅱ.①马…②麦…③考…④李… Ⅲ.单片微型计算机-微控制器 Ⅳ.TP368.1

中国版本图书馆CIP数据核字(2008)第111348号

内 容 提 要

本书是关于PIC微控制器的经典著作,内容紧密围绕PIC18系列微控制器原理及嵌入式系统应用展开,主要介绍了PIC18系列微控制器的硬件和软件方面的基本知识和特性,着重描述其硬件结构、软件编程和接口技术及其嵌入式应用等问题。

本书适合作为高等院校相关专业课程教材,也可供从事微控制器应用设计和嵌入式系统开发的工程技术人员参考。

图灵电子与电气工程丛书
PIC技术宝典

◆ 著 [美] Muhammad Ali Mazidi, Rolin D. McKinlay, Danny Causey
译 李中华 张雨浓 陈卓怡 等
责任编辑 朱 巍
◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子函件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京铭成印刷有限公司印刷
◆ 开本:700×1000 1/16
印张:45
字数:1067千字 2008年10月第1版
印数:1—4000册 2008年10月北京第1次印刷

著作权合同登记号 图字:01-2007-3148号
ISBN 978-7-115-18554-9/TP

定价:99.00元

读者服务热线:(010)88593802 印装质量热线:(010) 67129223
反盗版热线:(010)67171154

版 权 声 明



译 者 序

微控制器是一种无处不在的内嵌型控制芯片，广泛用于工业控制、消费类电子产品、汽车、金融、军事、能源等领域。在强大的市场需求和日新月异的科技进步驱动下，微控制器产品不断推陈出新。PIC 系列微控制器是全球领先的微控制器和模拟半导体供应商——美国微芯公司推出的嵌入式微控制器，具有运行速度快、工作电压低、功耗低、驱动能力强、体积小、价格低廉、指令简单易学易用等优点。PIC18 系列微控制器就是其中的杰出产品之一。

本书涵盖学习微控制器所需的基础知识、PIC18 微控制器的内部结构、汇编指令系统、C 语言编程、接口技术及其应用实例，具有以下鲜明特点。

- ☐ 本书采用计算机基础知识——PIC18 微控制器内部结构——汇编指令——C 编程——外围接口——应用实例的授课思路，知识体系清晰、内容丰富，适应当今信息社会对宽口径计算机类专业学生的培养需求。
- ☐ 每章的开篇都简明地给出了本章的学习目标以及主要内容的结构分布，在每章末尾还给出了小结，有助于读者进一步理解和思考所学知识，形成完整的知识体系。
- ☐ 本书在介绍 PIC18 微控制器原理及应用设计的同时，还配以丰富的设计实例，让读者能充分体会到每一个设计细节，有利于快速培养读者的细致分析与设计系统的能力。
- ☐ 对于主要的设计实例，本书分别给出了其汇编语言编程和 C 语言编程，通过对比学习两类不同的编程思想和风格，使得学生很容易发现二者的联系、区别和优缺点，克服了将汇编语言编程和 C 语言编程单独讲述的弊端。
- ☐ 章后习题以及每节后的复习题进一步充实了全书的内容，有利于帮助读者更好地理解相关基本理论。本书配有的附录，内容全面详尽，对读者而言是一本非常难得的技术手册。

简而言之，本书结构清晰、内容丰富、通俗易懂，是一本不可多得的学习宝典。

本书由李中华、陈卓怡主译，李中华负责统稿，张雨浓负责审校。参加本书翻译和初校工作的还有杨波、孙宇佳、张董华、伍沛然等。在此，谨对所有为本书的出版提供了帮助的人们表示诚挚的谢意。

由于译审者水平所限，加之时间仓促，译文中难免有不妥乃至错误之处，敬请广大读者批评指正。

译 者

前　　言

使用微处理器的产品大致可分为两类。第一类产品使用高性能微处理器，比如“奔腾”系列 CPU，应用在系统性能很重要的场合。第二类产品对性能的要求是次要的，价格、大小、功耗以及快速开发等问题比原始处理能力更为重要，用于这种产品的微处理器常称为微控制器。

本书介绍微控制器。PIC18 是目前广泛应用的一种微控制器。其应用范围广泛的原因有很多，Microchip 公司在软件和硬件上的大力支持是一个不可忽视的原因。本书可以用作微控制器和嵌入式系统的大学课程教材。它不仅建立了汇编语言程序设计的基础，而且针对工程学科的学生全面讲述了 PIC18 接口。基于这样的背景，进而讨论了基于微控制器的嵌入式系统的设计与接口。本书也可供应用技术人员、硬件工程师、计算机科研人员和爱好者使用。对于那些构建单机项目或者采用 PC 进行数据采集与传送的网络项目设计人员来说，本书也是一本理想的参考资料。

预备知识

读者应该具备数字电路相关知识。了解汇编语言相关知识有助于学习本书，但这些知识不是必需的。虽然本书是为那些没有汇编语言程序设计背景的学生而写的，但预先有汇编程序设计经验的学生将能够迅速掌握 PIC18 的体系结构，并且能立即开展项目。对于书中的 PIC18 C 语言程序设计部分，C 语言的基础知识是必需的。在书中，我们采用 Microchip 公司的 PIC18 C 语言编译器。PIC18 C 编译器与 MPLAB 相兼容，并且可以在 Microchip 公司网站(www.microchip.com)免费获得。我们鼓励读者用 MPLAB 仿真和运行书中的程序。

本书概览

本书采用系统的、循序渐进的叙述方式，涵盖了 PIC18 的 C 语言与汇编语言程序编程和接口的各个方面，还提供了许多实例和例程，用以澄清概念，并向学生提供动手学习的机会。在每节的结尾都配有习题，巩固本节的要点。

第 0 章介绍数制系统(二进制、十进制和十六进制)、基本逻辑门与计算机术语。这章专为学生设计，如没有学过数字逻辑课程的机械工程专业的学生，或者那些需要回顾这部分知识的学生。

第 1 章讨论了 PIC18 的历史和其他 PIC 系列产品(如 PIC16)的特点，还提供了 PIC18 系列部分芯片型号的一览表。

第 2 章讨论了 PIC18 的内部体系结构，介绍了如何用 PIC18 汇编语言创建可执行程序，

还研究了栈和标志寄存器。

第3章讨论了循环、跳转和调用指令，并给出了大量的例程。

第4章致力于讨论 I/O 端口。了解了这部分内容，学生可以开始做 PIC18 I/O 接口的实验，并且尽快启动项目。

第5章专注于算法、逻辑指令和程序的介绍。

第6章涉及 PIC18 的寻址方式，以及怎样访问 PIC18 程序存储区的数据和怎样进行存储器组转换。

第7章介绍 PIC18 的 C 语言程序设计。在整本书中，我们都使用美国 Microchip 公司提供的 PIC18 C 语言编译器编写 PIC18 及其他系列的 C 语言程序。PIC18 的 C 编译器与 MPLAB 兼容，并且可以在 Microchip 公司网站上免费获得。

第8章介绍 PIC18 芯片的硬件连接。

第9章描述了 PIC18 定时器以及怎样用它们作为事件计数器。

第10章研究了 PIC18 的串行数据通信及其同 RS232 的接口，还介绍了 PIC18 与 x86 IBM PC 及其兼容机的 COM 端口的通信。

第11章详尽讨论了 PIC18 中断，并附有诸多编写中断处理程序的例子。

第12章介绍了 PIC18 与实体设备(如液晶显示屏和键盘)的接口。

第13章介绍了 PIC18 与实体设备(如 DAC 芯片、ADC 芯片以及传感器)的接口。

第14章阐述了怎样用 PIC18 闪存和 EEPROM 存储器作为数据存储单元。

第15章介绍了 PIC18 内部的 CCP 和 ECCP 模块以及它们的用法。

第16章介绍了怎样使用 SPI 总线协议对实时时钟芯片 DS1306 进行连接和编程。

最后，第17章介绍了继电器、光绝缘体和电动机的基本接口。

附录给出了本书主题的所有必备参考资料。附录 A 详尽地描述了 PIC18 的每一条指令，并附有例子。附录 B 介绍了绕接技术的基本知识。附录 C 包含了 IC 技术和逻辑器件，以及 PIC18 的 I/O 端口接口和扇出能力。在将 PIC18 连接到外部设备之前必须学习这部分内容。附录 D 探究了流程图与伪代码的用法。附录 E 适用于那些熟悉 x86 和 8051 体系结构又想快速过渡到 PIC18 体系结构的学生。附录 F 为 ASCII 表。附录 G 列出了汇编共享软件的资源和一些电子器件供应商。附录 H 包含了 PIC18 芯片的数据表。

实验手册

实验手册包含了一些非常基本的实验，并且可以在 www.MicroDigitalED.com 网站上找到。实验指导教师可根据课程目标、授课层次和研究生课程或本科生课程，安排更加高级和严格的实验任务。相关材料和作者的其他书籍都可以在上述网站上找到。

习题答案和 PPT 教辅

章末习题涉及了一些很基本的概念。教师可以根据授课目标、授课层次和研究生课程或本科生课程，安排更有挑战性和更严格的任务。习题答案是在 Rasti 先生和 Faramarz

Mortezae 教授的帮助下写成的。只有教师才有资格获得在线的习题答案和 PPT 教辅[1]。

在线教师资源

如果要在线获得补充材料，教师必须获得一个教师登录密码。登录 www.prenhall.com，单击 Instructor Resource Center(教师资源中心)链接，然后单击 Register Today(现在注册)，获取教师登录密码。注册后的 48 小时之内，你将收到一封确认电子邮件，其中包含了一个教师登录密码。一旦获得教师登录密码，就可以下载想用的资料了。

致谢

本书是众人智慧和汗水的结晶。我们向所有对本书提供帮助的人表示诚挚的谢意。

感谢 Esfahan 大学的 Javad Rasti 先生。他在本书出版之前，仔细通读了每一个章节，发现并修正了一些错误。书中的大部分图表都是根据 Pedram Mazidi 的 PIC18 数据表再创作的。一些教授、专业工程师和学生发现了书中的错误，或者对书的改进提出了建议。我们真诚地感谢他们的热情与支持。他们是 Javad Rasti(Esfahan 大学)、Vahid Mokhtari(BIHE[2])、Mohammadi Abdar(Azad 大学)、Clyde Knight、Sam Waterman 和 Faramarz Mortezaei(都来自 DeVry 大学)、Frank Fortman、David Goodman 和 Maryam Mohseni。本书的出版离不开他们的鼓励。

感谢审阅这版书的专家和学者：

Shujen Chen，DeVry 大学 Tinley Park 校区；

Lawrence Lam，DeVry 大学 Federal Way 校区；

Vahid Mokhtari，BIHE 大学；

Faramarz Mortezaie，DeVry 大学 Fremont 校区；

Sepehr Naimi，BIHE 大学；

Javad Rasti，Esfahan 大学；

Chao-Yin Wang，DeVry 大学 North Brunswick 校区。

最后，感谢 Prentice Hall 的各位工作人员，尤其感谢在写作方面给了我们很大支持和鼓励的编辑 Jeff Riley 和促成本书出版的制作编辑 Rex Davidson。我们有幸得到了世界上最好的文稿编辑 Janice Mazidi 和 Bret Workman 的帮助，感谢他们所做的无与伦比的工作。

编写此书赋予了作者很多的乐趣，希望你在阅读时也能获得快乐，并将它运用在课程学习和项目设计中。如果你有任何建议或者发现书中存有错误遗漏，请与我们联系：

mdebooks@yahoo.com

---

① 如果您是高校教师，有意采用本书作为教材，请致信 contact@turingbook.com。——编者注

② BIHE 是 Bahái Institute for Higher Education 的缩写，这是一所伊朗的高等学府。——编者注

mmazidi @ microdigitaled. com
rmckinlay @ microdigitaled. com
dcausey @ microdigitaled. com

汇编器/编译器

MPLAB 和 PIC18 的 C 编译器可以在 http://www.microchip.com 网站下载。

商标信息声明

目　　录

第 0 章 计算入门

学习目标:

- ☐ 二进制、十进制、十六进制之间的相互转换
- ☐ 十六进制数的加减法
- ☐ 二进制数的加法
- ☐ 任意二进制数的补码
- ☐ 任意字符串的 ASCII 代码
- ☐ 逻辑操作 AND、OR、NOT、XOR、NAND 和 NOR
- ☐ 使用逻辑门的简单电路图
- ☐ 位、半字节、字节和字之间的区别
- ☐ 千字节、兆字节、吉字节和太字节的准确数学定义
- ☐ RAM 和 ROM 之间的区别以及它们的作用
- ☐ 计算机系统主要组成部分的功能
- ☐ 3 种计算机总线以及它们的作用
- ☐ 计算机系统里 CPU 的作用
- ☐ CPU 的主要部件以及它们的功能

1

为了理解基于微控制器系统的软件和硬件,首先必须掌握一些计算机设计中最基本的概念。本章(按数字计算机的传统称它为第 0 章)将介绍数制和编码系统的基础知识。在介绍完逻辑门后,会简要地描述计算机内部的运作。最后一节将介绍 CPU 结构的简史。虽然有些读者对本章的主题背景已经熟悉了,但还是建议读者浏览一下。

0.1 数制和编码系统

人类使用十进制算术,但计算机使用的是二进制系统。本节将解释十进制系统和二进制系统之间的相互转换;还会介绍二进制的一个简便的表示方法,即十六进制;最后会探究字符代码的二进制格式 ASCII 码。

0.1.1 十进制和二进制数制系统

虽然大多数人都推测十进制系统的采用是源于人类的十根手指,但计算机采用二进制系统背后的原因就不是那么明显了。计算机采用二进制系统是因为 1 和 0 分别代表开和关两个电平。十进制有 10 个独立的符号,0、1、2、……、9,而二进制只用 0 和 1 来组成数。十进制有 0

到9的数字，二进制只有数字0和1。这两个数字(0和1)通常叫作位(bit)。

0.1.2 十进制数转换成二进制数

把十进制数转换为二进制数的一个方法是，用十进制数不断除以2，记录余数，直至商为零，然后把余数按逆序来写，就得到二进制数。请看例0-1的演示。

例0-1 把25_{10}转换成二进制数。

解：

	商	余数	
25/2 =	12	1	LSB （最低位）
12/2 =	6	0	
6/2 =	3	0	
3/2 =	1	1	
1/2 =	0	1	MSB （最高位）

因此，$25_{10}=11001_2$。

0.1.3 二进制数转换成十进制数

要把二进制数转换成十进制数，首先要理解和每个数位有关的权(weight)的概念。首先，回想十进制里的权，如右侧表格所示。相似地，二进制里的每个数位都有相应的权：

740683_{10} =	
3×10^0 =	3
8×10^1 =	80
6×10^2 =	600
0×10^3 =	0000
4×10^4 =	40000
7×10^5 =	700000
	740683

110101_2 =			十进制	二进制
1×2^0	=	1×1 =	1	1
0×2^1	=	0×2 =	0	00
1×2^2	=	1×4 =	4	100
0×2^3	=	0×8 =	0	0000
1×2^4	=	1×16 =	16	10000
1×2^5	=	1×32 =	32	100000
			53	110101

知道了二进制数每一位的权，就可以容易把它们相加得到对应的十进制数，如例0-2所示。

例0-2 把11001_2转换成十进制数。

解：

权：	16	8	4	2	1
数：	1	1	0	0	1
和：	16+	8+	0+	0+	$1=25_{10}$

知道了二进制数每一位对应的权，就可以无需除法过程，把十进制数直接转换成二进制数。如例0-3所示。

例 0-3 使用权的概念把 39_{10} 转换成二进制数。

解：

权：	32	16	8	4	2	1
	1	0	0	1	1	1
	32+	0+	0+	4+	2+	1=39

因此，$39_{10}=100111_2$。

0.1.4 十六进制系统

十六进制，或者是计算机术语中的十六进制系统，是二进制数的一种简便表示。例如，相比于一大串的 0 和 1，它更易读，如 100010010110 和它的十六进制形式 896H。二进制系统有 2 个数字，0 和 1。十进制系统有 10 个数字，0 到 9。十六进制系统有 16 个数字。在十六进制中，前 10 个数字(0～9)和十进制的一样，剩下的 6 个数字使用的是字母 A、B、C、D、E 和 F。表 0-1 列出了用二进制、十进制和十六进制方法表示的 0～15。

表 0-1 十六进制系统

十进制	二进制	十六进制	十进制	二进制	十六进制
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	A
3	0011	3	11	1011	B
4	0100	4	12	1100	C
5	0101	5	13	1101	D
6	0110	6	14	1110	E
7	0111	7	15	1111	F

0.1.5 二进制和十六进制之间的转换

为了把二进制数表示成十六进制形式，首先要从右边开始，每 4 个位一次，把每 4 位二进制数用表 0-1 中的对应十六进制数代替。为了把十六进制数转换成二进制数，每一个十六进制数位要用 4 位对应的二进制数来代替。请看例 0-4 和例 0-5。

例 0-4 把二进制数 100111110101 用十六进制表示。

解：首先把该数分成每 4 位一组：1001 1111 0101。然后把每一组用对应的十六进制数代替：

1001	1111	0101
9	F	5

因此，$100111110101_2=9F5$(十六进制数)。

例 0-5 把十六进制数 29B 转换成二进制数。

解：

2 9 B

= 0010　1001　1011

把开头的 0 去掉，得到 1010011011。

0.1.6　十进制数转换成十六进制数

把十进制数转换成十六进制数有以下两种方法。

(1) 先转成二进制数，再转成十六进制数。例 0-6 说明这种十进制数转换成十六进制数的方法。

4

(2) 直接用除法取余数把十进制数转成十六进制数。这个方法的实现留给读者。

例 0-6

(a) 把 45_{10} 转成十六进制。

32	16	8	4	2	1	首先转成二进制。
1	0	1	1	0	1	32+8+4+1=45

$45_{10}=0010\ 1101_2$=2D(十六进制数)

(b) 把 629_{10} 转成十六进制。

512	256	128	64	32	16	8	4	2	1
1	0	0	1	1	1	0	1	0	1

$629_{10}=(512+64+32+16+4+1)=0010\ 0111\ 0101_2$=275(十六进制数)

(c) 把 1714_{10} 转成十六进制。

1024	512	256	128	64	32	16	8	4	2	1
1	1	0	1	0	1	1	0	0	1	0

$1714_{10}=(1024+512+128+32+16+2)=0110\ 1011\ 0010_2$=6B2(十六进制数)

0.1.7　十六进制数转换成十进制数

把十六进制数转换成十进制数也有两种方法。

(1) 把十六进制转成二进制，然后转成十进制。例 0-7 演示了这种十六进制数转成十进制数的方法。

(2) 直接用加权方法把十六进制数转成十进制数。

例 0-7　把下面的十六进制数转换成十进制数。

(a) $6B2_{16}=0110\ 1011\ 0010_2$

1024	512	256	128	64	32	16	8	4	2	1
1	1	0	1	0	1	1	0	0	1	0

$1024+512+128+32+16+2=1714_{10}$

(b) $9F2D_{16}=1001\ 1111\ 0010\ 1101_2$

32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
1	0	0	1	1	1	1	1	0	0	1	0	1	1	0	1

$32768+4096+2048+1024+512+256+32+8+4+1=40\ 749_{10}$

5

0.1.8 十进制、二进制和十六进制计数

为了说明这 3 种数制的关系，表 0-2 给出了 0 到 31 的十进制数序列和它们对应的二进制数和十六进制数。注意，在每一种数制中，当最大数码加 1 后，该位就会变成 0，并向高位进位 1。例如，十进制中，9+1=0，向高位进 1。二进制中，1+1=0，带一个进位；相似地，在十六进制中 F+1=0，带一个进位。

表 0-2 数制计数

十进制	二进制	十六进制	十进制	二进制	十六进制
0	00000	0	16	10000	10
1	00001	1	17	10001	11
2	00010	2	18	10010	12
3	00011	3	19	10011	13
4	00100	4	20	10100	14
5	00101	5	21	10101	15
6	00110	6	22	10110	16
7	00111	7	23	10111	17
8	01000	8	24	11000	18
9	01001	9	25	11001	19
10	01010	A	26	11010	1A
11	01011	B	27	11011	1B
12	01100	C	28	11100	1C
13	01101	D	29	11101	1D
14	01110	E	30	11110	1E
15	01111	F	31	11111	1F

0.1.9 二进制数和十六进制数加法

二进制数的加法是一个很直观的过程。表 0-3 说明了两数的加法。这里忽略二进制数的减法，因为所有计算机都是用加法完成减法的。虽然计算机有加法器电路，但是没有独立的减法器电路，取而代之的是使用加法器配合补码电路执行减法。换句话说，为了完成$(x-y)$，计算机先对 y 取补，然后和 x 相加。补码的概念会在下面介绍。例 0-8 介绍了二进制数的加法。

表 0-3 二进制加法

A+B	进位	和
0+0	0	0
0+1	0	1
1+0	0	1
1+1	1	0

例 0-8　对下面的二进制数进行相加。用对应的十进制数检查你的答案。

解：

二进制	十进制
1101	13
+ 1001	9
10110	22

0.1.10　补码

要得到二进制数的补码，就要把所有位取反，然后对结果加 1。取反就是简单地把全部 0 变成 1、1 变成 0。请看例 0-9。

例 0-9　对 10011101 取补。

解：

10011101	二进制数
01100010	取反
+ 1	
01100011	取补

0.1.11　十六进制数的加法和减法

在学习计算机的软件和硬件相关知识的时候，通常需要进行十六进制数的加减法。掌握这些方法是很必要的。下面将分别介绍十六进制数的加法和减法。

0.1.12　十六进制数的加法

本节将描述十六进制数的加法处理。从最低位开始，所有的数相加，如果结果小于 16，那么写出该位的结果；如果大于 16，那么将结果先减去 16 得到差，再向高位进 1。最好使用例子来解释这种方法，如例 0-10 所示。

例 0-10　计算十六进制数加法：23D9＋94BE。

解：

23D9	最低位：9＋14＝23	23－16＝7 带进位
+ 94BE	1＋13＋11＝25	25－16＝9 带进位
B897	1＋3＋4＝8	
	最高位：2＋9＝B	

0.1.13　十六进制数的减法

两个十六进制数相减，如果减数大于被减数，需要向高位借 16。请看例 0-11。

例 0-11 计算十六进制数减法:59F−2B8。

解:

```
  59F          最低位:15−8=7
− 2B8          25(9+16)−11=14(E)
  2E7          4(5−1)−2=2
```

0.1.14 ASCII 码

到目前为止讨论的都是数字系统的表示方法。在计算机中的所有信息都是用 0 和 1 表示的,所以二进制数也用来表示字母和其他字符。在 20 世纪 60 年代建立了一个标准的表示法,即 ASCII(American Standard Code for Information Interchange,美国信息交换标准码)。ASCII(读作 ask-E)码用二进制数表示了数字 0 到 9、所有的英文字母(包括大写和小写)以及很多控制代码和标点符号。这个系统最大的优点是绝大部分的计算机都是使用它的,因此信息可以相互共享。ASCII 码系统使用 7 位来表示每个代码。例如,100 0001用于表示大写字母 A,而 110 0001 用于表示小写字母 a。通常在最高位补 0 把它作为一个 8 位代码。图 0-1 列出了部分 ASCII 码。附录 F 中有完整的 ASCII 代码表。ASCII 码的使用不仅为美国和很多其他国家的键盘提供了一个标准,而且为打印机和显示器等输出设备的打印和显示字符确定了标准。

十六进制数	符号	十六进制数	符号
41	A	61	a
42	B	62	b
43	C	63	c
44	D	64	d
...	...	...	...
59	Y	79	y
5A	Z	7A	z

图 0-1 部分 ASCII 码

注意,发明 ASCII 码是为了操作 ASCII 数据简便。例如,数码 0 到 9 用 ASCII 码 30 到 39 表示。这样,程序只需要把 ASCII 高半字节的 3 屏蔽,就能轻松得到其十进制数。还要注意的是,大写字母和小写字母是有对应关系的。ASCII 码用 41 到 5A 来表示大写字母,而用 61 到 7A 表示小写字母。注意二进制代码,大写字母 A 和小写字母 a 只有第 5 位是不同的。因此,简单改变 ASCII 码的第 5 位就能实现大小字母的转换。

0.1.15 复习题

1. 为什么计算机用的是二进制系统而不是十进制系统呢?
2. 把 34_{10}转换成二进制数和十六进制数。
3. 把 110101_2转换成十进制数和十六进制数。
4. 执行二进制加法:101100+101。
5. 把 101100_2 转换成它的补码形式。
6. 计算 36BH+F6H。
7. 计算 36BH−F6H。
8. 写出 80x86 CPU 的 ASCII 码(十六进制形式)。

0.2　数字入门

本节将概括性地介绍数字逻辑和设计。首先将介绍二进制逻辑操作;接着介绍执行这些功能的逻辑门;然后是如何使用逻辑门组合成简单的数字电路;最后,将介绍微控制器接口常用的一些逻辑器件。

0.2.1　二进制逻辑

如前面提到的,计算机使用二进制系统,因为两个电平可以用数码0和1表示。数字电子学中的信号有两个独立的电平。例如,一个系统可以定义0 V为逻辑0,定义+5 V为逻辑1。图0-2画出了这个带内部电平偏差容限的系统。这个例子的有效数字信号应该是落在图中的其中一个阴影部分。

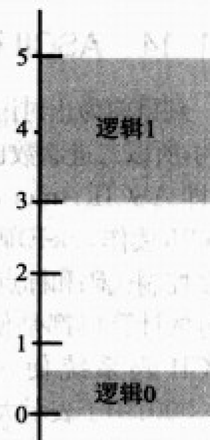


图0-2　二进制信号

0.2.2　逻辑门

二进制逻辑门是这样一种简单的电路,它接收一个或多个输入信号,然后发送一个输出信号。下面将对一些逻辑门进行定义。

1. 与门

与门接收两个或两个以上输入,然后对它们执行逻辑与运算。请看下面的与门的真值表和图形符号。注意,如果与门的两个输入都是1,那么输出就为1。其他任何的输入组合得到的输出都为0。例子用了两个输入,*X*和*Y*。逻辑门也可能有多路输出。对于与门,如果全部输入为1,输出就为1。如果任何一个输入为0,那么输出就为0。

2. 或门

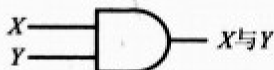
逻辑或的特性是,如果有一个或一个以上的输入为1,那么输出就为1。当且仅当全部输入都为0时,输出才为0。

3. 三态缓冲

缓冲门并不改变输入的逻辑电平。它用来隔离或者放大信号。

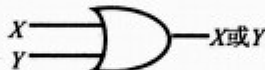
逻辑与特性

输入	输出
XY	*X*与*Y*
0 0	0
0 1	0
1 0	0
1 1	1

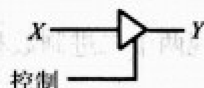


逻辑或特性

输入	输出
XY	*X*或*Y*
0 0	0
0 1	1
1 0	1
1 1	1



缓冲



4. 反相器

反相器也被称为非门，输出是输入值的取反。也就是，输入为1，输出为0；而输入为0，输出为1。

5. 异或门

异或门执行的是输入的异或操作。当输入有且仅有一个1时，异或产生的输出为1。如果两个操作数都为0，输出为0。如果两个操作数都为1，输出也为0。注意，从异或门的真值表可以看出，当两个输入相同时，输出就为0。这个特性可以用来比较两个位是否相同。

逻辑反相器

输入	输出
X	非X
0	1
1	0



逻辑异或特性

输入	输出
XY	X异或Y
0 0	0
0 1	1
1 0	1
1 1	0

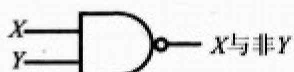


6. 与非门和或非门

与非门的特性就像是一个与门在输出端加上一个非门。当输入都为1时，输出为0；反之，输出为1。或非门的特性就像是一个或门在输出端加上一个非门。当所有输入都为0时，输出为1；反之，输出为0。因为与非门和或非门的生产工艺简单而且成本低，所以常用于数字设计。任何使用与门、或门、异或门以及非门设计的电路，都可以转换成只使用与非门和或非门。下面给出这样的一个简单的例子。注意，对于与非门，只要有一个输入为0，输出就为1。而对于或非门，只要有一个输入为1，输出就为0。

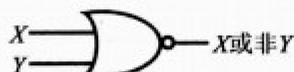
逻辑与非特性

输入	输出
XY	X与非Y
0 0	1
0 1	1
1 0	1
1 1	0



逻辑或非特性

输入	输出
XY	X或非Y
0 0	1
0 1	0
1 0	0
1 1	0



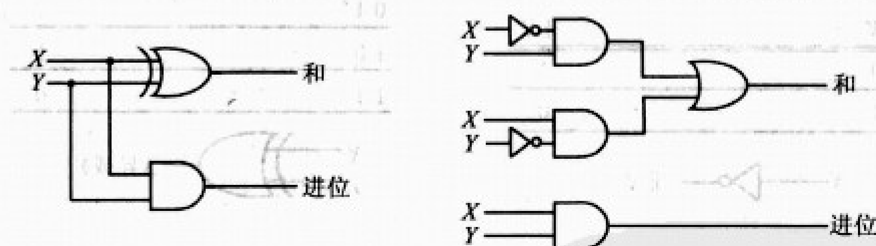
0.2.3 使用逻辑门设计电路

下面介绍用一个简单的逻辑设计实现两个二进制数相加。两个1位二进制数加法有下面4种结果：

	进位	和
0+0=	0	0
0+1=	0	1
1+0=	0	1
1+1=	1	0

要注意，当计算(1+1)时，得到的结果为0，带进位。在这个设计中，需要确定和以及进位。可以注意到，上面求和栏的值对应异或(XOR)函数的输出，而进位栏的值对应与(AND)函数的输出。图0-3a画出了用异或门和与门组成的简单加法器。图0-3b画出了使用与门、或门和反相器组合而成的同功能电路。

图0-4画出了半加器的模块图。两个半加器可以组合成能执行3个输入的加法器，称为全加器。图0-5画出了全加器的逻辑电路，还有屏蔽了细节的模块图。图0-6画出了使用3个全加器的三位加法器。



(a) 使用异或门和与门的半加器　　(b) 使用与门、或门、反相器的半加器

图0-3　两种半加器



图0-4　半加器的模块图

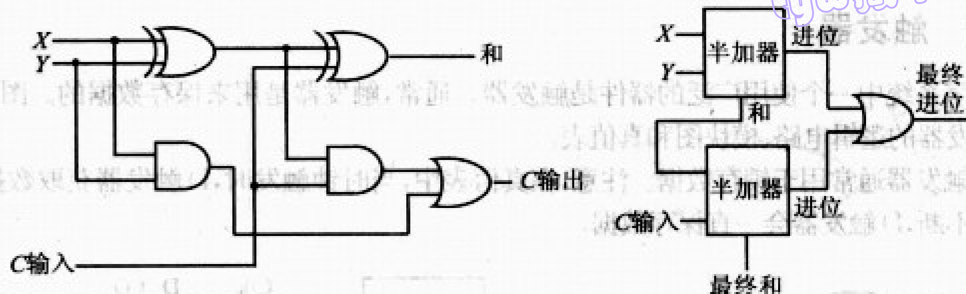


图 0-5 半加器组成全加器

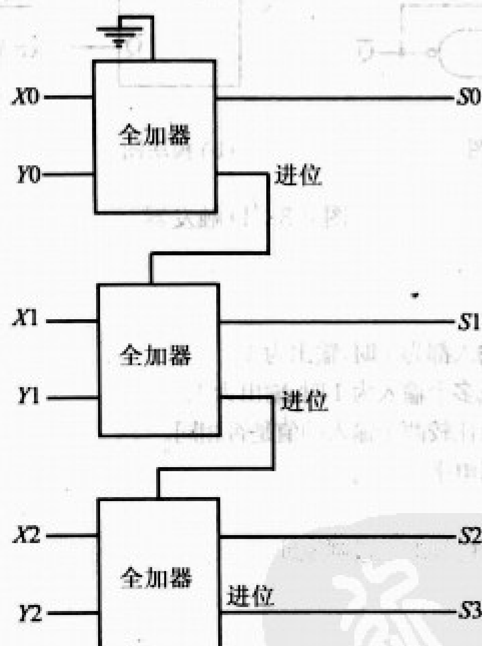
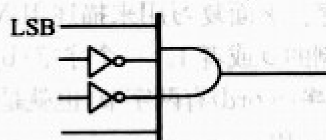


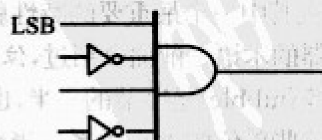
图 0-6 使用 3 个半加器的三位加法器

0.2.4 译码器

逻辑门的另一个应用例子就是译码器。译码器广泛用于计算机设计中的地址译码。图 0-7 画出了使用反相器和与门的 9(1001,二进制)和 5(0101)的译码器。



(a) 9(二进制，1001)的译码器。当且仅当输入是二进制数1001时，与门的输出为1



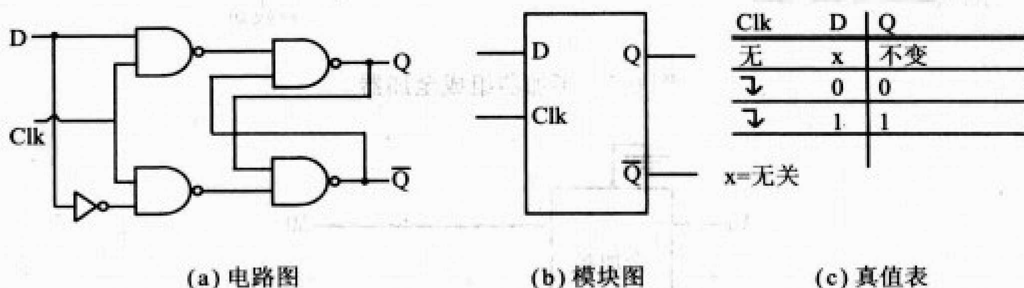
(b) 5(二进制，0101)的译码器。当且仅当输入是二进制数0101时，与门的输出为1

图 0-7 地址译码器

0.2.5 触发器

数字系统中一个使用广泛的器件是触发器。通常，触发器是用来保存数据的。图0-8画出了触发器的逻辑电路、模块图和真值表。

D触发器通常用于锁存数据。注意，在真值表中，当时钟触发时，D触发器获取数据。只要电源不断，D触发器会一直保存数据。



(a) 电路图 (b) 模块图 (c) 真值表

图0-8 D触发器

0.2.6 复习题

1. 逻辑操作________在所有输入都为1时，输出为1。
2. 逻辑操作________在一个或多个输入为1时，输出为1。
3. 逻辑操作________通常用来比较两个输入的值是否相同。
4. ________不改变输入的逻辑电平。
5. 请说出触发器的一个用途。
6. 地址________用来确认一个已知的二进制地址。

0.3 计算机内部

本节将介绍计算机的组成和内部工作。这里使用的是一般模型，但谈到的概念适用于所有的计算机，包括IBM PC、PS/2和兼容机型。在开始这个话题前，有必要先回顾一些最常用到的计算机术语，如K(千)、M(兆)、G(吉)、字节、ROM、RAM等。

0.3.1 一些重要术语

计算机其中一个最重要的特性就是它有多少内存。下面复习用来描述IBM PC和兼容机型存储器的术语。前面介绍过，位(bit)可以是二进制的0或者1。一个字节(byte)定义为8位。半字节(nibble)是字节的一半，也就是4位。一个字(word)有两字节，也就是16位。右表中画出了这些单位的大小关系。当然，它们都可以是0和1的任意组合。

Bit	0
Nibble	0000
Byte	0000 0000
Word	0000 0000 0000 0000

千字节(kilobyte)是2^{10}字节，也就是1024字节。字母KB是kilobyte的缩写。例如，有些软盘可容纳365KB数

据。兆字节(megabyte,有时也写为 meg)是 2^{20}字节。它比一百万字节稍微大一点;准确地说是 1 048 576 字节。更大的单位有:吉字节(gigabyte)是 2^{30}字节(超过十亿),太字节(terabyte)是 2^{40}字节(超过一万亿)。为了说明这些术语的一些用法,假设一台计算机有 16MB 的内存;这可以表示为 16×2^{20},或者 $2^4\times2^{20}$,也就是 2^{24}。因此 16MB 也就是 2^{24}字节。

微型计算机里常用的内存有两种:RAM,也就是随机访问存储器(Random Access Memory)(有时候也叫作读/写存储器);ROM,也就是只读存储器。RAM 通常用作计算机程序运行时的临时存储单元。当计算机关闭后,数据就会丢失,因此,RAM 有时也叫作易失存储器。ROM 里存储了计算机的程序和基本操作信息。ROM 里面的数据是永久性的,不能被使用者改变,在电源关闭后也不会丢失,因此,它也叫作永久性存储器。

13

0.3.2 计算机的内部组成

每台计算机的内部工作都可以分成三部分:CPU(中央处理器)、内存和 I/O(输入/输出)设备(如图 0-9 所示)。CPU 的功能是执行(处理)内存中的信息。I/O 设备(如键盘和显示器)提供了与 CPU 通信的工具。CPU 通过总线(bus)连接到内存和 I/O 设备。计算机内部的总线把数据从一个地方送到另一个地方,就如同大路上的公共汽车把人从一个地方送到另一个地方。每台计算机都有 3 类总线:地址总线、数据总线和控制总线。

一个设备(内存或者 I/O)要能被 CPU 识别,就必须被分配一个地址。分配给已知设备的地址必须是唯一的;不同的设备不能有相同的地址。CPU 把地址(肯定是二进制形式)送入地址总线,经译码电路找出设备。然后 CPU 通过数据总线从设备获得数据或者是发送数据到设备。控制总线是用来向设备发送读/写信号的,说明 CPU 现在是等待接收信息还是发送信息。在这 3 种总线中,地址总线和数据总线决定了 CPU 的性能。

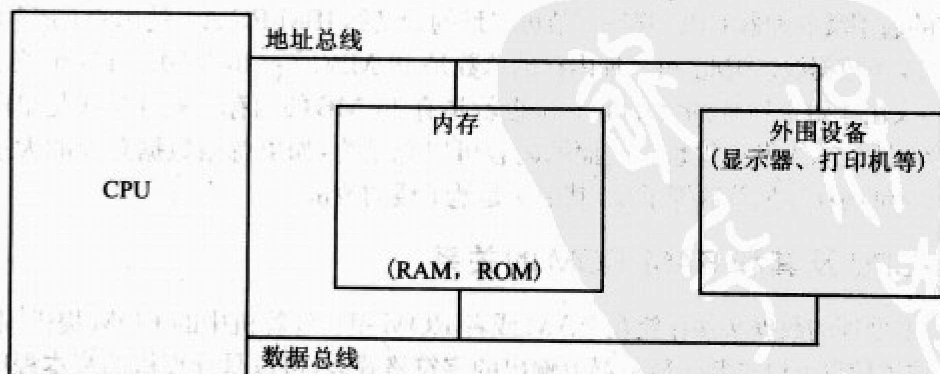


图 0-9 计算机内部

0.3.3 数据总线概述

因为数据总线是 CPU 用来接收或发送信息的,所以数据总线越多,CPU 性能越好。如果把数据总线比喻成高速公路,那么显然车道越多,CPU 和其他设备(如打印机、ROM、RAM 等;如图 0-10 所示)之间的联络就越畅通。相似地,增加车道的数目会增加建筑成本。越多的数据总线意味着 CPU 和计算机的成本越高。不同 CPU 的数据总线通常是 8 位或者 64 位。

早期的个人计算机(如苹果 2)使用的是 8 位数据总线，而巨型机(如 Cray)使用的是 64 位数据总线。数据总线是双向传输的，因为 CPU 要用它们来接收和发送数据。计算机的处理能力和总线的大小有关，因为 8 位总线每次能发送 1B 数据，而 16 位总线每次能发送 2 B 数据，速度增加了 1 倍。

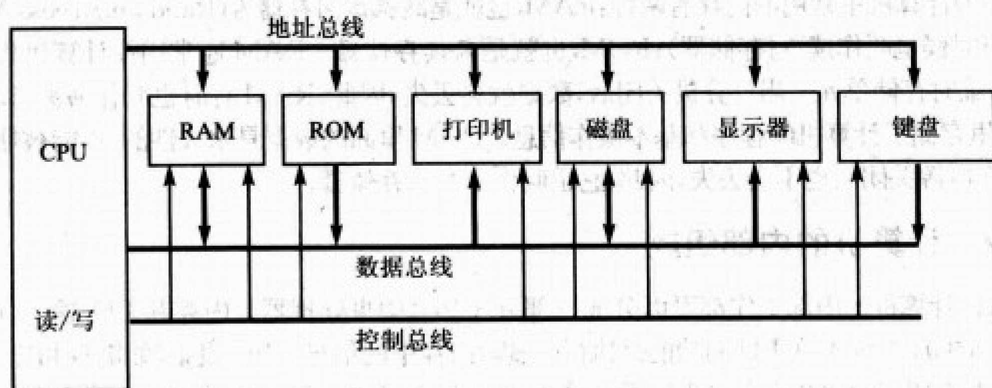


图 0-10　计算机内部组成

14

0.3.4　地址总线概述

因为地址总线是用于识别连接到 CPU 的设备和内存的，所以地址总线越多，能分配地址的设备数目就越多。换句话说，CPU 的地址总线数目决定它可用的地址数量。如果忽略数据总线的大小，地址的数量总等于 2^x，其中 x 是地址总线数。假设一个 CPU 有 16 条地址总线，它可以提供 65 536(2^{16})B(或者说是 64 KB)的可寻址内存。每个地址最多有 1 B 的数据。这是因为所有的通用微处理器 CPU 都是字节可寻址的。又如 IBM PC AT 使用的 CPU 有 24 条地址线和 16 条数据线。因此，可寻址内存的总数是 16 MB(2^{24}＝16 MB)。在这个例子中，有 2^{24} 个地址，又由于每个地址的大小是 1 B，也就是有 16 MB 的内存。地址总线是单向总线，CPU 只能向地址总线发送地址。上面的内容可以总结为：如果忽略数据总线的大小，那么 CPU 的可寻址内存地址总数等于 2^x，其中 x 是地址线的数量。

0.3.5　CPU 及其和 RAM、ROM 的关系

CPU 要处理的数据必须存放在 RAM 或者 ROM 里。计算机中的 ROM 提供固定和永久信息。这类信息可以是要在显示器上输出的字符格式表，可以是计算机的基本程序(如测试和计算系统的 RAM 大小)，也可以是在显示器上输出信息。相反，RAM 存储的是可随时间变化的临时信息，如不同版本的操作系统和应用程序(如文字处理或者税率计算程序)。这些程序从硬盘驱动器读到 RAM 供 CPU 处理。由于硬盘的速度太慢，CPU 不能直接从硬盘读取信息。换句话说，CPU 先从 RAM(或者是 ROM)里寻找要处理的信息。只有当 CPU 需要的数据不在内存里的时候，CPU 才会从更大的存储空间(如磁盘)里寻找，然后把该信息送到 RAM。因此，RAM 和 ROM 有时又被称作*初级存储器*，而磁盘被称作*二级存储器*。图 0-11画出了 PC 内部组成的模块图。

15

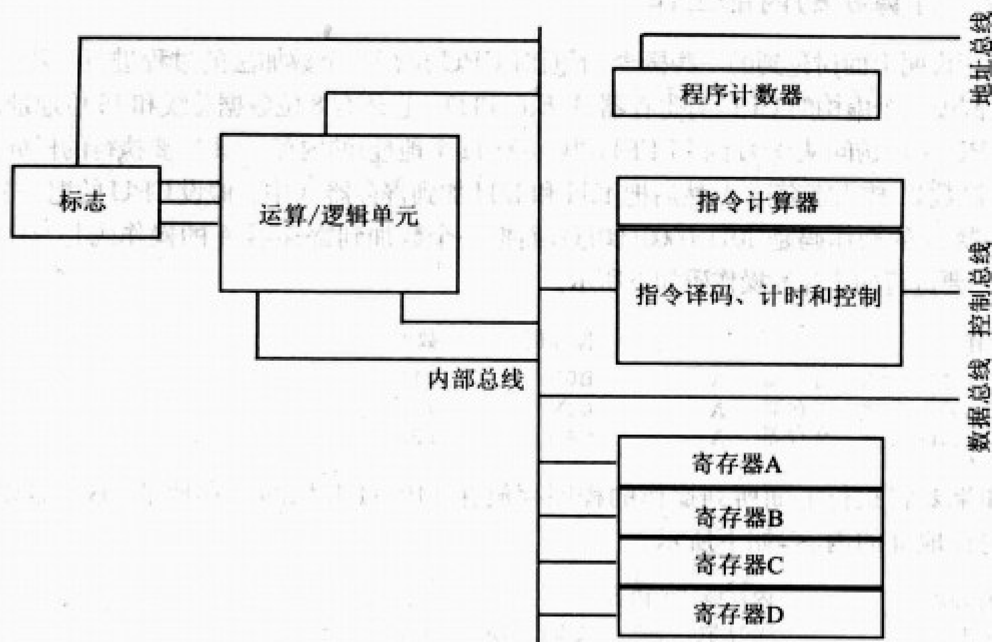


图 0-11 CPU 的内部模块图

0.3.6 CPU 内部

内存里的程序向 CPU 提供操作指令。操作可以是简单的数据相加(如工资单),也可以是一个机器的控制(如机器人)。CPU 的功能就是获取并执行内存里的指令。为了完成获取和执行,所有的 CPU 都需要配置以下资源。

(1) CPU 最重要的配置是一定数量的寄存器。CPU 靠寄存器来存储临时信息。信息可能是要处理的两个数,或者是要从内存里读取的数据的地址。CPU 里的寄存器可以是 8 位、16 位、32 位甚至是 64 位的,CPU 不同,寄存器的位数就不同。通常,寄存器越多、越大,CPU 的性能就越好。不过这也有一个缺点,就是 CPU 的成本也会相应变高。

(2) CPU 还需要 ALU(运算/逻辑单元)。CPU 的 ALU 部分负责执行运算功能(如加法、减法、乘法和除法)以及逻辑功能(如与、或、非)。

16

(3) 每个 CPU 都有一个程序计数器。程序计数器的功能是指向下一条要执行的指令的地址。当每条指令被执行后,程序计数器就会自加 1,指向要执行的下一指令地址。程序计数器的内容是送入地址总线来找出和获取目标指令的。在 IBM PC 里,程序计数器是一个叫作 IP 的寄存器,或者是指令指针。

(4) 指令译码器的功能是对 CPU 获得的指令进行解释。可以把指令译码器想象成一种字典,里面有每条指令的意思以及 CPU 对接收指令的操作步骤。正如字典都需要很多页来存放它定义的文字一样,一个 CPU 要能识别更多的指令,它就需要配备更多的晶体管。

0.3.7 计算机的内部工作

为了说明上面讨论到的一些概念，下面对CPU执行3个数加法的过程进行一步一步的分析。假设一个虚构的CPU有寄存器A、B、C和D。它还有8位数据总线和16位地址总线。因此CPU可以访问从0000到FFFFH(共10000H个地址)的内存。CPU要执行的任务是，把十六进制数21放入寄存器A，然后把42H和12H加到寄存器A中。假设CPU的把一个数移入寄存器A的操作码是1011 0000(B0H)，而把一个数加到寄存器A的操作码是0000 0100(04H)。要进行的步骤和操作码如下所示：

操作	操作码	数值
把21H 送入 寄存器 A	B0H	21H
把42H 加到 寄存器 A	04H	42H
把12H 加到 寄存器 A	04H	12H

如果要将执行上面所列操作的程序存放在1400H开始的内存地址，那么需要列出每个内存地址的内容，如下所示：

内存地址	内存地址的内容
1400	(B0)把一个数送入寄存器A的操作码
1401	(21)要移动的数
1402	(04)把一个数加到寄存器A的操作码
1403	(42)要加上的数
1404	(04)把一个数加到寄存器A的操作码
1405	(12)要加上的数
1406	(F4)停止的操作码

CPU运行上面程序时的操作如下所示。

(1) CPU的程序计数器的值可以是0000到FFFFH中的任一个。此处的程序计数器必须设为1400H，说明第一条执行指令的地址。当程序计数器得到了第一条指令的地址时，CPU就准备要执行了。

17

(2) CPU把1400H放入地址总线，然后发送出去。CPU触发READ信号时，它告诉内存它需要的字节在地址1400H，内存电路就找到该地址。内存地址1400H的内容(B0)通过数据总线被送入CPU。

(3) CPU通过指令译码器得到指令B0的解释。当得到了该指令的定义之后，它知道它需要把下一个内存地址的字节送入CPU的寄存器A中。因此，它要求控制电路能严格执行。当从内存地址1401取得21H时，它需要保证除寄存器A以外的寄存器都关闭了。因此，当21H进入CPU时，它会直接进入寄存器A。完成一条指令之后，程序计数器指向下一个要执行指令的地址，在这里也就是1402H。地址1402被送到地址总线以获取下一条指令。

(4) CPU从内存地址1402H获得操作代码04H。译码后，CPU知道它要把下一地址(1403)的字节加入寄存器A。CPU把该数值(在这里是42H)送入寄存器A，寄存器A本身的内容和这个数就被送到ALU执行加法运算。CPU从ALU的输出端得到加法的结果并把它放入寄存器A。同时，程序计数器变成1404，也就是下一条指令的地址。

(5) 地址 1404H 被送入地址总线，得到操作码送入 CPU，译码，然后执行。这个还是把一个数加入寄存器 A 的代码。程序计数器更新为 1406H。

(6) 最后，地址 1406 的内容被获取和执行。HALT 指令告诉 CPU 停止对程序计数器加 1 并请求下一条指令。如果没有 HALT，CPU 会继续更新程序计数器并获取指令。

现在假设地址 1403 的值是 04，而不是 42H。那么 CPU 怎样识别是要加的数值 04 还是操作码 04 呢？记住，操作码 04 是告诉 CPU“把下一个值送入寄存器 A”。因此，CPU 不会对下一个值进行译码。它只是简单地把下面内存地址的值送入寄存器 A，而不会考虑该值大小。

0.3.8 复习题

1. 24 KB 有多少个字节？
2. RAM 的意思是什么？它在计算机系统中的作用是什么？
3. ROM 的意思是什么？它在计算机系统中的作用是什么？
4. 为什么 RAM 又叫作易失存储器？
5. 说出计算机系统的 3 个主要部分。
6. CPU 的意思是什么？说明它在计算机中的功能。
7. 说出计算机系统中的 3 种总线，并且简要介绍每一种总线的用途。
8. 指出下面的总线哪个是单向的，哪个是双向的。
   (a) 数据总线　(b) 地址总线
9. 如果一台计算机的地址总线有 16 位，它最大的可寻址内存大小是多少？
10. ALU 的意思是什么？它的作用是什么？
11. 计算机系统中的寄存器有什么用处？
12. 程序计数器的功能是什么？
13. 指令译码器的功能是什么？

18

小结

二进制系统的所有数都是由 0 和 1 两个数码组合而成的。二进制系统的使用对于数字计算机来说是非常重要的，因为只有两种状态需要表示：开和关。任何二进制数都可以直接转换成人们更易读的十六进制数。从二进制/十六进制转换成十进制，又或者是反过程，经过练习以后都会变得非常的直观和简单。ASCII 码是计算机内部用来表示字符型数据的二进制代码。它通常用于外围设备的输入和输出。

逻辑与门、或门和反相器是构筑简单电路的基本模块。与非门、或非门和异或门同样可用于电路的设计。在讲授如何用逻辑门进行电路设计的时候，文中列举了半加器和全加器的例子。译码器是用来识别已知地址的。触发器是用在其他电路就绪前锁存数据的。

任何计算机系统的主要组成部分包括：CPU、内存和 I/O 设备。“内存”是指临时或者永久数据的存储器。在大部分的系统中，内存可以按字节访问，也可以按字访问。术语千字节、兆字节、吉字节和太字节是用来描述大量字节的单位。计算机系统中的内存主要有两种：

RAM和ROM。RAM(随机访问存储器)用于临时存放的程序和数据;ROM(只读存储器)用于永久存放计算机系统实现最基本功能的程序和数据。计算机系统的所有部件都由CPU控制。外围设备(如I/O设备)让计算机可以与人以及其他计算机进行交流。计算机中有3种总线:地址、控制和数据。控制总线是CPU用来控制其他设备的;地址总线是CPU用来确定设备或者内存地址的;数据总线用于CPU和其他设备间的数据发送和接收。

19

最后,本章还概述了数字逻辑。

习题

1. 把下面的十进制数转换成二进制数:
   (a)12 (b)123 (c)63 (d)128 (e)1000
2. 把下面的二进制数转换成十进制数:
   (a)100100 (b)1000001 (c)11101 (d)1010 (e)00100010
3. 把第2题中的数转换成十六进制。
4. 把下面的十六进制数转换成二进制数和十进制数:
   (a)2B9H (b)F44H (c)912H (d)2BH (e)FFFFH
5. 把第1题中的数转换成十六进制。
6. 找出下面二进制数的补码:
   (a)1001010 (b)111001 (c)10000010 (d)111110001
7. 计算下面的十六进制加法:
   (a)2CH+3FH (b)F34H+5D6H
   (c)20000H+12FFH (d)FFFFH+2222H
8. 计算下面的十六进制减法:
   (a)24FH−129H (b)FE9H−5CCH
   (c)2FFFFH−FFFFFH (d)9FF25H−4DD99H
9. 说出数字0、1、2、3、……、9的ASCII码、十六进制和二进制形式。
10. 说出下面字符串的ASCII码(十六进制形式):
    “U. S. A is a country” CR,LF
    “in North America” CR,LF
    (CR表示回车,LF表示换行)
11. 用2输入的或门画出一个3输入的或门。
12. 写出3输入或门的真值表。
13. 用带2个输入的与门画出一个3输入的与门。
14. 写出3输入与门的真值表。
15. 用2输入的异或门设计一个3输入的异或门,并写出3输入的异或门的真值表。
16. 写出3输入与非门的真值表。
17. 写出3输入或非门的真值表。
18. 画出二进制码1100的译码器。
19. 画出二进制码11011的译码器。
20. 写出D触发器的真值表。

21. 回答下面的问题：
    (a) 16 位的半字节是多少？
    (b) 32 位的半字节是多少？
    (c) 如果一个字定义为 16 位，那么 64 位数据段有多少个字？
    (d) 1 MB 的准确空间是多少？用十进制表示。
    (e) 多少 KB 为 1 MB？
    (f) 1 GB 的准确空间是多少？用十进制表示。
    (g) 多少 KB 为 1 GB？
    (h) 多少 MB 为 1 GB？
    (i) 如果一个系统有 8 MB 的内存，那是多少字节（十进制表示）？又是多少千字节呢？

20

22. 一个已知的存储设备（例如硬盘）可存放 2GB 的信息。假设一个文档的每一页有 25 行，每一行有 80 列的 ASCII 字符（每个字符＝1B），那么这个硬盘大约可以存放多少页这样的信息呢？
23. 已知一个字节可寻址计算机，内存地址从 1000H 到 9FFFFH 可以存放用户程序。首地址是 10000H，尾地址是 9FFFFH。计算下面各值。
    (a) 可用字节的总数（十进制形式）。
    (b) 可用千字节的总数（十进制形式）。
24. 一台计算机有一条 32 位数据总线。那么每次最多可以有多少信息可以送入 CPU 呢？
25. 下面列出了一些计算机和其数据总线大小。列出每个 CPU 一次能接收的最大数据量（用十六进制和十进制表示）。
    (a) Apple 2，8 位数据总线。
    (b) IBM PS/2，16 位数据总线。
    (c) IBM PS/2 的 80 型机，32 位数据总线。
    (d) Cray 超级计算机，64 位数据总线。
26. 根据下面 CPU 的地址总线大小，找出对应的内存大小，用所给单位表示。
    (a) 16 位地址总线（KB）。
    (b) 24 位地址总线（MB）。
    (c) 32 位地址总线（MB 和 GB）。
    (d) 48 位地址总线（MB、GB、TB）。
27. 数据总线和地址总线，哪个是单向的哪个是双向的？
28. CPU 的哪个寄存器是用来存放指令地址的？
29. CPU 的哪一部分负责执行加法？
30. 说出每个 CPU 里的 3 种总线类型。

21

复习题答案

0.1 节

1. 计算机使用二进制系统，是因为每一位有两个电平：开和关。
2. $34_{10}=100010_2=22_{16}$　3. $110101_2=35_{16}=53_{10}$　4. 1110001　5. 010100　6. 461　7. 275
8. 38 30 78 38 36 20 43 50 55 73

0.2 节

1. 与 2. 或 3. 异或 4. 缓冲器 5. 存储数据 6. 译码器

0.3 节

1. 24 576
2. 随机访问存储器;它用来临时存放CPU正在执行的程序,如操作系统、文字处理程序等。
3. 只读存储器;它用来存放永久程序,如键盘控制程序等。
4. RAM的内容会在计算机断电后丢失。
5. CPU、内存和I/O设备。
6. 中央处理器;它可以当成是计算机的"大脑";它执行程序和计算机的所有其他设备。
7. 地址总线传输CPU需要的地址;数据总线传输进出CPU的信息;控制总线是CPU用来发送I/O设备控制信号的。
8. (a) 双向 (b) 单向
9. 64 KB,或者是65 536 B。
10. 运算/逻辑单元;它执行所有的算术和逻辑操作。
11. 它们用于临时存放信息。
12. 它保存下一条要执行的指令的地址。
13. 它告诉CPU每一条指令的执行步骤。

第1章 PIC 微控制器的历史和特性

学习目标：

- ☐ 微处理器和微控制器的区别
- ☐ 微控制器在一些应用中的优势
- ☐ 嵌入式系统的概念
- ☐ 微控制器的选用标准
- ☐ 微控制器的运算速度、封装形式、内存容量和单片价格，以及这些因素对微控制器选择的影响
- ☐ PIC 系列产品之间的比较
- ☐ PIC 同其他厂家生产的微控制器之间的比较

下面先讨论微控制器在日常生活中的角色及其重要性。1.1 节讨论选择微控制器的准则及其在嵌入式市场中的应用；1.2 节涵盖了 PIC18 系列丰富多样的成员。另外，本章还简略探讨了 PIC 芯片的替代者（如 8051、AVR 及 68HCC11 微控制器）。

1.1 微控制器与嵌入式处理器

这一节讨论微处理器的需求，并把它们和通用目的微处理器（如奔腾以及其他 X86 微处理器）做一个比较。本章还介绍了微控制器在嵌入式市场里所扮演的角色。此外，本章提供了一些如何挑选微控制器的准则。

1.1.1 微控制器和通用微处理器

微控制器和微处理器的区别究竟是什么？微处理器指的是通用目的微处理器，如 Intel 公司的 x86 家族（8086、80286、80386、80486 和奔腾）或者摩托罗拉公司的 PowerPC 系列。这些微处理器本身的芯片上并没有 RAM、ROM 和 I/O 端口，因此，它们通常被称作通用微处理器，如图 1-1 所示。

一名系统设计师在使用像奔腾或者 PowerPC 这样的通用微处理器时，必须从外部加入 RAM、ROM、I/O 端口和定时器来使整个系统运作正常。尽管外加的这些 RAM、ROM 和I/O 端口使得这些系统变得体积庞大并且造价高昂，它们仍然有着多功能的价值，这使得设计者在决定这些适合特定工作所需要的 RAM、ROM 和 I/O 端口总量的时候变得方便。然而这种

情况却不适用于微控制器。微控制器在一块芯片上集成了一个 CPU(微处理器)以及固定数量的 RAM、ROM、I/O 端口和定时器。换句话说，处理器、RAM、ROM、I/O 端口和定时器都被嵌入到了一块芯片上，因此，设计者不能对其外加任何外部存储器、I/O 端口或者定时器。由于芯片上 ROM、RAM 和 I/O 端口的数量是固定的，这就使得微控制器成为许多对造价和空间有着严格要求的应用系统的理想选择。许多应用(如远程电视操控)是不需要 486 甚至是 8086 的计算能力的。很多实际应用中，空间占用、能量消耗和单片价格都是较计算能力更为重要的考虑因素。这些实际应用通常需要 I/O 操作来读取信号、启动或关闭位信号。因此，这类处理器又被称为 IBP(Itty-Bitty Processor)。(参考 Rick Grehan 的 *Good Things in Small Packages Are Generating Big Product Opportunities*，摘自 *BYTE* 杂志，1994 年 9 月号。可登录 http://www.byte.com 获取关于微控制器的精彩论述。)

24

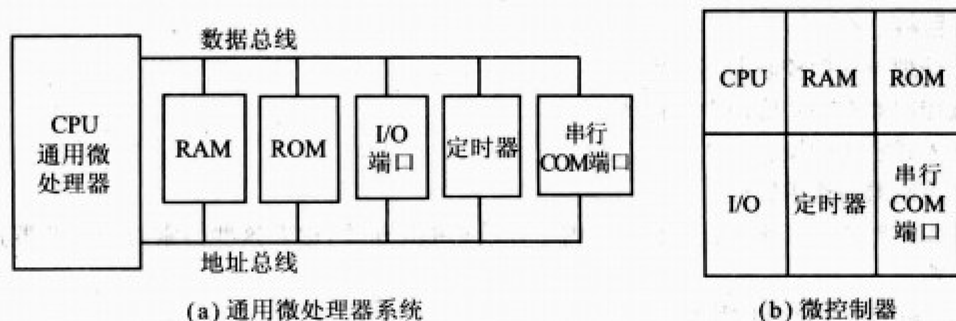


图 1-1　微处理器系统和微控制器系统的比较

有趣的是，一些微控制器的生产公司已经将 ADC(模数转换器)和其他外围设备整合到它们的微控制器里。

1.1.2　应用于嵌入式系统的微控制器

当在学术层面讨论微处理器的时候，我们经常会遇到一个术语：嵌入式系统。微处理器和微控制器被广泛应用在嵌入式系统的产品中。一个嵌入式产品由内部自带的微处理器(或微控制器)而不是外部控制器控制。典型的嵌入式系统的微控制器的 ROM 是为系统的特定功能而烧制的。例如，打印机就是嵌入式系统的一个例子。它的处理器只执行一种功能，也就是获取和打印数据。与之相反，基于奔腾处理器的个人电脑(或者任何 x86 系列 IBM 兼容的个人电脑)通常具有很多功能，例如文字处理、打印服务、银行出纳、电脑游戏、网络服务，又或者是因特网终端。个人电脑还可以加载和运行多种应用软件。当然，个人电脑可以执行庞大任务的一个原因是，它拥有 RAM 和操作系统，可以把应用软件加载到 RAM 并由 CPU 运行。而对于一个嵌入式系统，只有单一的应用软件被烧制到 ROM 内。一台 x86 个人电脑本身已经包括或者连接了多种嵌入式产品，例如键盘、打印机、调制解调器、磁盘控制器、声卡、CD-ROM、鼠标等。每一种外围设备都有一个微控制器负责执行唯一的功能。例如，在每一个鼠标里，微控制器负责捕捉鼠标的位置并将其传输给电脑。表 1-1 列出了一些含嵌入式芯片的产品。

表 1-1 一些使用微控制器的嵌入式产品

家电		办公室	汽车
厨房电器	远程控制装置	电话	掌上电脑
内部对讲机	视频游戏机	电脑	发动机控制器
电话	手机	安全防盗系统	安全气囊
安全防温系统	电子乐器	传真机	防滑刹车系统
车库门开启器	缝纫机	微波炉	车用仪表
留言机	照明控制器	复印机	安全系统
传真机	传呼机	激光打印机	传动控制
家用电脑	相机	彩色打印机	娱乐设备
电视机	弹球机	传呼机	温度控制
有线电视遥控器	电子玩具		车载电话
录像机	健身设备		遥控门锁
便携摄像机			

1.1.3 x86PC 嵌入式应用

虽然微控制器是很多嵌入式系统的首选，但是有时一个微控制器并不是对所有任务都适用的。因此，近年来，很多通用微处理器的制造商[例如 Intel、Freescale Semiconductor(以前的 Motorola)以及 AMD(Advanced Micro Device)]都瞄准了高端嵌入式微处理器市场。Intel 和 AMD 大力向嵌入式和台式电脑市场推出它们的 x86 处理器。在 20 世纪 90 年代初，苹果电脑开始使用 PowerPC 微处理器(604、603、620 等)来取代以往使用的 680X0 系列。[①] PowerPC 微处理器是 IBM 和 Motorola 公司的一次成功合作，而且瞄准了高端嵌入式和个人电脑市场。要注意的是，当一个公司锁定一种通用微处理器为嵌入式市场目标后，它会尽最大努力优化它的产品以适应嵌入式系统。因此，这些处理器通常称为*高端嵌入式处理器*。另外，ARM 微处理器也被广泛用于高端嵌入式系统设计。经常提及的术语*嵌入式处理器*和*微控制器*是可以互相替换的。

降低功耗和占用空间，是评价嵌入式系统的一个重要标准。当 CPU 芯片上集成更多功能时，便可达到这个要求。所有基于 x86 和 PowerPC 6xx 的嵌入式处理器都具有低功耗的特点，而且在一块芯片上还集成了 I/O、COM 端口和 ROM。对于高性能嵌入式处理器，在 CPU 芯片中集成更多功能，供设计者根据需要选择使用，这正成为一种趋势。这种趋势也向着个人电脑系统设计蔓延。通常在设计电脑主板的时候，我们需要一个 CPU 和一个包含了 I/O、高速缓存控制器、带有 BIOS 的闪存 ROM、二级高速缓存的芯片集。新的设计在工业上不断涌现。例如，很多公司都推出了这样一种芯片，它包含了完整的 CPU，还有除 DRAM 外的逻辑支持电路和存储器。可以说，这样的一块芯片就是一台电脑。

目前，由于 Linux、MS-DOS 和 Windows 标准的存在，许多嵌入式系统都采用 x86 个人电脑模式。很多情况下，使用 x86 的个人电脑模式的高端嵌入式设计不仅节省资金，还因为丰富

25

① 最近苹果电脑也开始采用 Intel 芯片。——编者注

的 Linux、DOS 和 Windows 软件库而缩短开发时间。事实上，因为 Windows 和 Linux 有广泛的使用，而且平台为人所熟悉，基于 Windows 和 Linux 的嵌入式产品的开发会大大节省资金和开发时间。

1.1.4 微控制器的选择

目前有5种主流的8位微处理器。它们是：Freescale Semiconductor(过去的 Motorola)的68HC08/68HC11，Intel 的8051，Atmel 的 AVR，Zilog 的 Z8，还有 Microchip 公司的 PIC。[①] 以上每一种微处理器都有独特的指令集和寄存器集，因此，它们是彼此不兼容的，为其中一种处理器编写的程序不能在其他处理器上运行。除此以外，还有很多芯片厂家制造16位、32位微处理器。在这么多的微处理器中，设计者根据什么样的准则选择呢？以下是要考虑的3点标准：(1)能有效和经济地解决手头的计算任务；(2)具有软硬件工具的适用性，如编译、汇编、调试和仿真；(3)具有丰富的可用性和可靠的微处理器资源。接下来，让我们详细介绍以上标准。

26

1.1.5 微控制器的选择标准

(1) 选择一个微控制器的首要标准是，它能有效完成手头的任务并且成本经济。在分析一个以微控制器为基础的项目的需求时，我们首先要判断：在8位、16位和32位的微控制器中，谁能最有效地处理当前的计算任务。除此还有其他的考虑因素。

(a) 速度。微控制器能支持的最高运算速度是多少。

(b) 封装。它是一个有40个引脚的双列直插式封装(DIP)、一个方形扁平式封装(QFP)，还是其他的封装形式？这一点对于最终产品的空间大小、组装以及成型都很重要。

(c) 功耗。这一点要求对于电池供电的产品尤为重要。

(d) 芯片上 RAM 和 ROM 的总数。

(e) 芯片的 I/O 引脚以及定时器数量。

(f) 是否易于升级到更高性能或者更低功耗的版本。

(g) 单片价格。在最后产品的成本计算中，这一点显得很重要。例如，有些微控制器一次购买100 000片时，单价仅50美分。

(2) 第二个标准是围绕选用的微控制器我们是否容易开发产品。主要的考虑包括汇编器、调试器、代码高效的C语言编译器、仿真器等的可用性以及专业内外的技术支持。在很多情况下，第三方卖主(例如一个供应商而非芯片制造商)对芯片的维护，即使不比生产商做得更好，也能提供同等的服务。

(3) 第三项标准就是它是否已经具备满足在目前及未来实际需求量的能力。有些设计者会认为这一点比前两点更重要。目前，主流的8位微控制器8051系列已经拥有数量最大的多样化(多源)供应商(供应商是指除了微控制器开发者之外的制造商)。除了最先开发8051的 Intel 公司，目前不少公司也在生产(或者曾经生产过)8051系列芯片。

值得注意的是，由于 Freescale Semiconductor、Atmel、Zilog 和 Microchip 公司等公司产品具有稳定、成熟和单一的特点，所以它们已经投入大量人力、财力去巩固产品的应用广泛性

---

① 事实上，ARM 处理器也是非常流行的选择。——编者注

和时效性。近年来，很多公司开始出售针对不同微控制器的现场可编程门阵列（FPGA）和专用集成电路（ASIC）库。

1.1.6 机电学与微控制器

在一个逐渐兴起的领域——机电学里，微控制器扮演着主要的角色。以下是摘取自澳大利亚纽卡斯尔大学网页（http://mechatronics2004.newcastle.edu.au/mech2004）关于机电学研究领域最精彩的总结，该大学每年都会举行一次重要的机电学年会。

27

"在机电学和电子工程领域中的很多技术程序和产品，表明机械学正和电子学、信息处理技术结合得越来越紧密。这种由元件（硬件）和信息驱动功能（软件）相结合构成的系统，就是所谓的机电学系统。

"机电学系统的发展包括如何在基础机械结构、传感器和执行器实现、数字信息自动处理器和全局控制之间找到最佳平衡的过程，而这些协调得益于技术革新。机电学的实际应用需要涵盖众多学科的专业知识，如机械工程、电子学、信息技术和决策理论。"

1.1.7 复习题

1. 判断正误：微控制器通常比微处理器便宜。
2. 比较一个基于微控制器的系统和一个通用微处理器。哪个更便宜？
3. 一个微控制器的芯片上通常配有以下哪个器件？
   (a) RAM　(b) ROM　(c) I/O　(d) 以上所有
4. 一个通用微处理器通常需要以下哪个器件附带配合使用？
   (a) RAM　(b) ROM　(c) I/O　(d) 以上所有
5. 嵌入式系统为什么又称为专用系统？
6. 嵌入式系统的概念是什么？
7. 为什么为给定产品提供多源是重要的？

1.2 PIC18 系列概述

本节首先纵观 PIC 系列微控制器，然后考察更多 PIC18 系列的细节。

1.2.1 PIC 微控制器的发展简史

1989 年，Microchip 公司开发了一种 8 位的微控制器：PIC(Peripheral Interface Controller)，也就是外围接口控制器。这种微控制器有少量的数据 RAM，为编程而设的数百字节的片上 ROM，一个定时器，还有一些 I/O 端口的引脚，而这些全部集成在一片只有 8 个引脚的芯片上（见图 1-2）。一个公司能在不到 10 年时间内把这么一种简陋的产品发展成主流的 8 位微控制器，实在是一个奇迹。在编写本书期间，Microchip 公司是世界上首屈一指的 8 位微控制器供应商。自从推出了 PIC16xxx，该公司开发的一系列 8 位微控制器已经在这里无法枚举了，其中包括 PIC 系列的 10xxx、12xxx、14xxx、16xxx、17xxx 和 18xxx。它们全都是 8 位处理器，也就是说 CPU 一次只能处理 8 位数据。如果超过了 8 位，就必须把数据分解成 8 位，以便 CPU 处理。PIC 系列存在的一个问题是，当从一个系列转到另一个系列的时候，它们并不是

百分之一百地做到软件升级的兼容。例如，12xxx/16xxx 有 12 位和 14 位宽的指令，而 18xxx 的指令是 16 位宽的，并且有许多新增指令；所以，当要在 18xxx 上运行为 12xxx 编写的指令时，我们在读入程序前必须重新编译程序并且可能要更改寄存器地址。在编写本书的时候，PIC18xxx 系列是 8 位 PIC 微控制器中性能最好的。事实上，PIC18xxx 对 18～80 个引脚的适用性，使它成为创新设计的理想选择。原因是，它能方便地移植到更强版本的芯片上而不会丧失了软件兼容性。当时，PIC18xxx 并没有现成的 8 引脚版本，所以，如果要设计所谓的小插件，需要选择 10xxx～16xxx 其他系列的产品。本书主要介绍 PIC18 系列，对于其他系列只介绍一些主要特性，希望读者能到 Microchip 的网站上去了解更多 10xxx～16xxx 系列的信息。如果已经熟练掌握 PIC18 系列，你会更容易和直接地认识其他系列。下面简要描述 PIC18 系列。

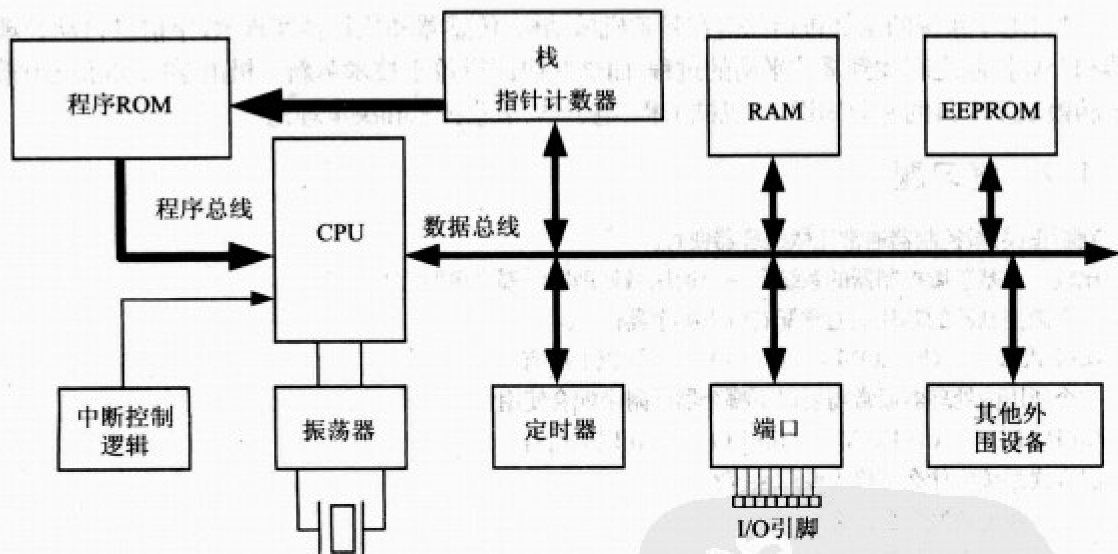


图 1-2　PIC 微控制器简图

1.2.2　PIC18 特性

PIC18 具有 RISC 体系结构和其他标准特征，如片上程序(代码)ROM、数据 RAM、数据 EEPROM、定时器、ADC、USART 和 I/O 端口。如图 1-2 所示。虽然该系列不同型号芯片在程序 ROM、数据 RAM、数据 EEPROM 和 I/O 端口的大小上不尽相同，但是他们都有定时器、ADC 和 USART 等外围设备。如图 1-3 和图 1-4 所示。因为这些外围设备的重要性，我们将专门开辟一章对它们进行详细描述。PIC18 的 RAM/ROM 存储器和 I/O 特性的详细介绍将在后面几章中阐述。

1. PIC 微控制器的程序 ROM

在微控制器中，ROM 是用来存储程序的，因此又被称作程序或代码存储器。虽然，PIC18 系列带有 2 MB 的程序 ROM 空间，但并不是所有 PIC 芯片都带有这样的存储器。在本书编写期间，程序 ROM 的大小取决于芯片型号，从 4 KB 到 128 KB 不等。PIC18 系列的程序 ROM 的类别也有几种，如闪存、OTP 和掩模 ROM。如果想了解更多的存储器技术，请阅读

第 14 章，那里会详细介绍各种类型的 ROM。注意，虽然 PIC18 系列的 RAM/ROM 在速度和数量上存有不同的配置，但对于本书提及的指令，它们都是适用的。换句话说，为一种芯片编写的程序也可以在其他芯片上运行。下面简要讨论 PIC18 系列的程序 ROM。

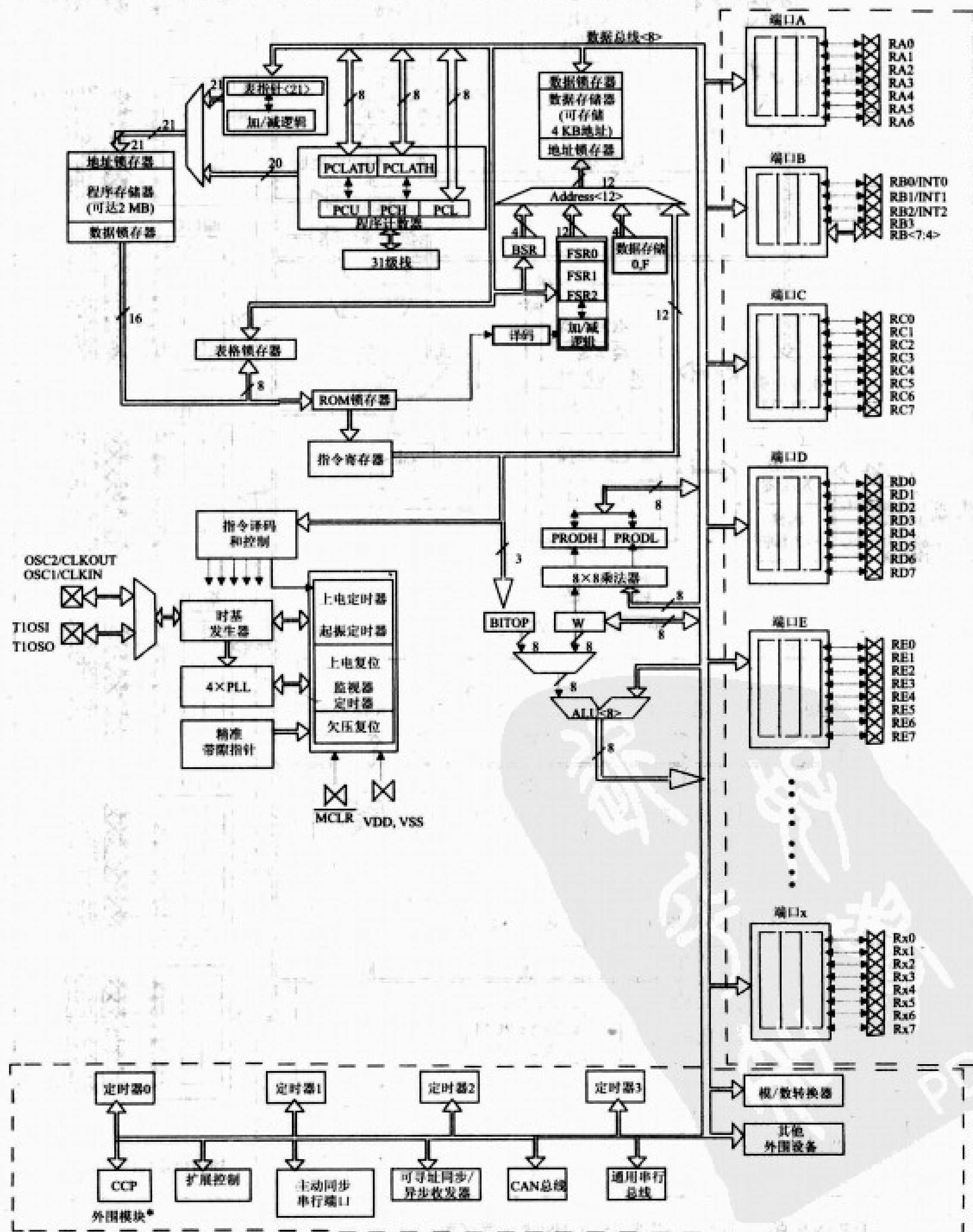
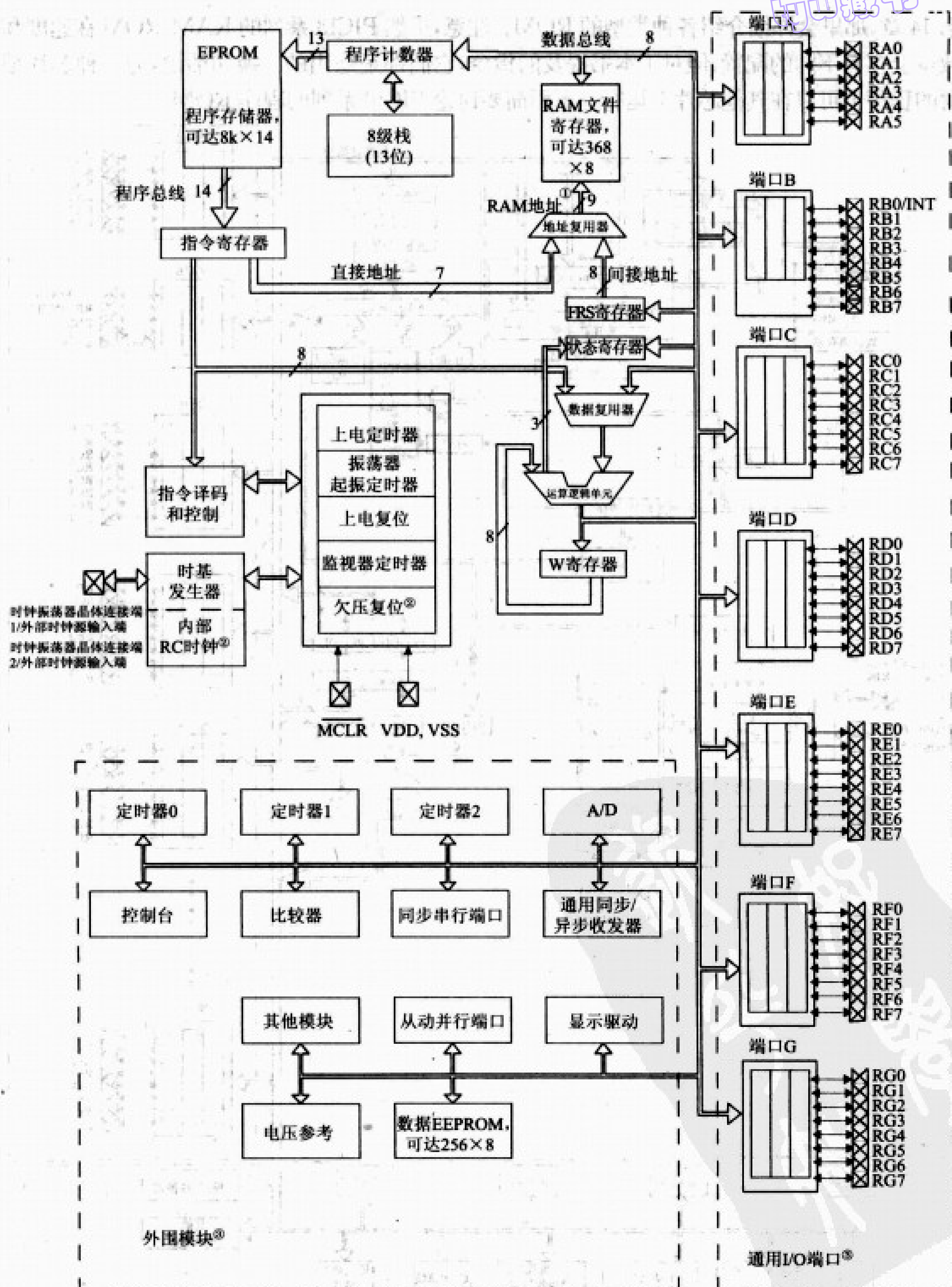


图 1-3 PIC18 框图



① RAM中直接地址的更高位来自STATUS(状态)寄存器。
② 不是所有设备都具有这个功能。请参阅设备的数据表。
③ 很多通用I/O引脚可复用为一个或者更多的外围模块功能，而多路复用功能和设备有关。

图 1-4　PIC16 框图

2. 带 UV-EPROM 的 PIC 微控制器

有些 PIC 微控制器使用 UV-EPROM(紫外线可擦除编程存储器)作为程序 ROM。使用这种芯片时,需要配合 PROM 烧录器和 UV-EPROM 擦除器,以擦除 ROM 的内容。紫外线从 UV-EPROM 上的窗口进入并擦除 ROM 上的数据。在使用这种存储器时值得注意的是,在数据需要等大约 20 min 擦除后才能重新写入数据。因此,Microchip 公司引入了闪存技术。目前,闪存正逐步地将 UV-EPROM 全部取代。表 1-2 描述了部分 PIC18 系列芯片。

表 1-2 PIC18 系列的部分芯片(http://www.microchip.com)

型 号	程序存储器大小	数据存储器大小	数据 EEPROM 大小	I/O 引脚数目	模数转换器大小	定时器数	引脚数和封装
PIC18F1220	4 KB(闪存)	256	256	16	10 位	4	18 DIP
PIC18F2420	16 KB(闪存)	768	0	25	10 位	4	28 DIP
PIC18F2220	4 KB(闪存)	512	256	25	10 位	4	28 DIP
PIC18F452	32 KB(闪存)	1536	256	34	10 位	4	40 DIP
PIC18F4520	32 KB(闪存)	1536	256	36	10 位	4	40 DIP
PIC18F458	32 KB(闪存)	1536	256	34	10 位	4	40 DIP
PIC18F4580	32 KB(闪存)	1536	256	36	10 位	4	40 DIP
PIC18F8722	128 KB(闪存)	3936	1024	70	10 位	5	80 TQFP

注意:
(1) 所有的 ROM、RAM 和 EEPROM 存储器的单位都为字节(byte)。
(2) 数据 RAM(通用 RAM)的大小是指 RAM 上用于数据操作的存储空间再加上 SFR 的大小。
(3) 以上所有芯片都采用通用同步/异步收发器作为串行数据传输。

3. 带闪存的 PIC18Fxxx

很多 PIC18 芯片都采用闪存作为片上程序 ROM。带有闪存的芯片的型号都以字母 F 作为区分。PIC18F458 就是典型的使用闪存的 PIC18 芯片。相比于 UV-EPROM 需要 20 min 甚至更多的时间来擦写,闪存的只需几秒钟,适合快速的开发需要。因此,PIC18F 正逐渐替代 UV-EPROM,可减少芯片擦除时间并加速开发时间。采用 PIC18F 的微控制器系统需要有支持闪存的 ROM 烧录器;但并不需要 ROM 擦除器,因为闪存是 EEPROM(electrically erasable PROM,电可擦除可编程存储器)。值得注意的是,要重新编程前,必须把 ROM 的所有内容都删除。这个过程是由 ROM 的程序编制器自身完成的,所以不需要外加擦除器。也可以借助 IBM 个人电脑的 USB 接口,将 Microchip 的 PICkit 2 连上 PIC18F 来编程。

4. PIC 的 OTP 版本

PIC 的 OTP(one-time-programmable,一次可编程)技术也是由 Microchip 公司提出的。PIC16C432 就是采用了 OTP 的程序 ROM。对比 PIC16C432 和 PIC28F252,字母 C 用来表示 OTP ROM,而 F 表示闪存。使用闪存的芯片更多是用于产品开发的,而当产品已经设计完善时,则多采用 OTP 的芯片进行大规模生产,因为后者的价格更经济。不过,OTP 产品中的程序无法再更改。

5. PIC 的掩模版

Microchip 公司可提供这样一种服务：把你的程序交给他们，他们会在生产过程中把你的程序烧录到 PIC 芯片上。这种芯片通常被称为"掩模 PIC"。同样，这也是 IC 制造的一个阶段。如果存储单元足够，那么掩模 PIC 是所有类别中最便宜的。因为掩模 PIC 加工没有最小的订单数量。

6. PIC 微控制器的数据 RAM 和 EEPROM

如果说 ROM 是用来存储程序(代码)的，那么 RAM 就是用来存储数据的。PIC18 最大的数据 RAM 容量为 4 KB。但不是所有的 PIC18 芯片都有如此大的 RAM 容量。PIC18 的 RAM 容量从 256 B 到 4096 B 不等。在下一章你将会看到，数据 RAM 主要包括通用 RAM (GPR)和特殊功能寄存器(SFR)。因为 SFR 是固定的而且每块芯片都必须有，所以实际上是 GPR 的容量导致数据 RAM 的容量区别如此之大。因此，Microchip 公司网站上只给出了 GPR 的容量。RAM 上 GPR 的存储空间，每 256 B 被分为一组，用于读/写数据的缓存与操作，这将在第 6 章中详细介绍。因此，PIC18 的 GPR 的大小总是 256 的整数倍。一些 PIC18 芯片还会采用一个小容量的 EEPROM 来存储重要而不需要经常修改的数据。用于数据缓存的 RAM 是每个 PIC18 芯片必备的，而 EEPROM 则是可选的，不是所有型号的 PIC18 芯片都有的。EEPROM 主要用于存储重要的数据，这将在第 14 章介绍。

33

7. 微控制器的 I/O 引脚

PIC18 芯片有数量为 16 到 72 个不等的 I/O 引脚。引脚的数目取决于芯片自身的封装形式。目前，PIC18 的引脚数目从 18 到 80 不等。例如 PIC18F1220 一共有 18 个引脚，其中 16 个用于 I/O；而 PIC18F8722 共有 80 个引脚，其中 72 个用于 I/O。第 4 章将介绍 I/O 引脚功能及编程技术。

8. PIC 微控制器的外围设备

PIC18 的所有芯片都以模数转换器、定时器、通用同步/异步收发器作为标准的外围设备。在第 13 章中将介绍，模数转换器是 10 位的，通道数目从 5 到 16 不等(取决于封装中的引脚数)。PIC18 最多可以有包括监视器定时器在内的 4 个定时器。第 9 章将详细论述定时器。通用同步/异步收发器用于将基于 PIC18 的系统连接到串行端口，如 IBM 个人计算机的 COM 端口，第 10 章将更深入地介绍这些内容。大部分的 PIC18 芯片都带有 I^2C 总线和 CAN 总线。

9. PIC Trainer

第 8 章将会广泛探讨 PIC18F458 Trainer 的设计。该 Trainer 通过 PICkit2 工具进行编程。MDEPIC Trainer 同 Microchip 公司的其他 40 引脚设备是兼容的。

1.2.3 其他微控制器

除 PIC 芯片外，市面上还是有很多广泛使用的 8 位微控制器，比如：8051 系列、68HC11 系列、AVR 系列和 Z8 系列。除 Intel 公司外，还有不少公司在生产 8051 芯片，如表 1-4 所示。AVR 系列是 Atmel 公司的产品，68HC11 系列及其改进型是 Freescale 公司的产品，Zilog 公司则是 Z8 的制造商。表 1-3 对比了 PIC18 系列和 8051/52 系列的特性。

表 1-3 8051 系列与 PIC18 系列(40 引脚封装)的比较

特 征	8051/52	PIC18xxx
程序 ROM(最大容量)	64 KB	2 MB
数据 RAM(最大容量)	256 B	4 KB
定时器	3	4
I/O 引脚	32	33
串行端口	1	1

34

表 1-4 生产 8 位微控制器的著名制造商

公 司	网 站	型 号
Microchip	http://www.microchip.com	PIC16xxx/18xxx
Intel	http://www.intel.com/design/mcs51	8051
Atmel	http:/www.atmel.com	AVR 和 8051
Philips/Signetics	http://www.semiconductors.philips.com	8051
Zilog	http:/www.zilog.com	Z8 和 Z80
Dallas Semi/Maxim	http://www.maxim-ic.com	8051
Freescale	http://www.freescale.com	68HC11/68HC08

完整数据见 http://www.microcontroller.com

关于 PIC 微控制器和 PIC Trainer 的详情,请登录以下网址:
http://www.microchip.com
http://www.MicroDigitalEd.com

1.2.4 复习题

1. 请描述 PIC8xxx 芯片的 3 个特征。
2. PIC18Fxxx 和 PIC18Cxxx 微控制器的主要区别是什么?
3. 请给出下面芯片的 RAM 容量大小:
   (a) PIC18F2420 (b) PIC18F4520
4. 请给出下面芯片的片上程序 ROM 容量大小:
   (a) PIC18F2420 (b) PIC18F4520
5. PIC18 系列是 ________ 位的微处理器。

小结

本章介绍了微控制器在日常生活中扮演的角色和重要作用。微处理器与微控制器既有区别也有联系。在讨论微控制器在嵌入式市场的应用的同时,还讨论了微控制器的选用标准,如运算速度、存储容量、封装形式和单片价格。1.2 节描述了 PIC 芯片的各系列(如 PIC18 和 PIC16)及其特性。此外,本章还讨论了 PIC18 系列的各类产品,如 PIC18F252 和 PIC18F458。

35

习题

1. 判断对错:通用微处理器含有片上 ROM。
2. 判断对错:通常,微控制器含有片上 ROM。
3. 判断对错:微控制器有片上 I/O 端口。
4. 判断对错:微控制器有固定数量的片上 RAM。
5. 微控制器片上通常包含哪些组件?
6. Intel 公司的 Pentium 芯片在 Windows 个人计算机上运行,需要外部的________和________芯片来存储数据和代码。
7. 举例列出可连接至个人计算机的 3 种嵌入式产品。
8. 请给出人们采用 x86 作为嵌入式处理器的理由。
9. 请给出一些最广泛使用的 8 位微控制器的型号和制造商。
10. 在第 9 题中,生产规模最大的是哪个制造商?
11. 对于一个采用电池供电的嵌入式产品,在选用微控制器时考虑的首要因素是什么?
12. 对于一个带有片上 ROM 的嵌入式控制器,为什么说 ROM 的容量大小很重要?
13. 在选用微控制器时,芯片的多源性有多重要呢?
14. 术语"第三方支持"如何理解?
15. 假设一个微控制器有 8 位和 16 位两个版本。下面的哪个描述是正确的?
    (a) 8 位的软件可以在 16 位的系统上运行。
    (b) 16 位的软件可以在 8 位的系统上运行。
16. PIC18F458 带有________字节的片上程序 ROM。
17. PIC18F2420 带有________字节的片上数据 RAM。
18. PIC18F452 带有________个片上定时器。
19. PIC18F458 有________字节的片上数据 RAM。
20. 浏览 Microchip 的网站,看看是否有无 ROM 的 PIC18 芯片。如果有,请给出它的型号名。
21. PIC18F458 带有________个 I/O 引脚。
22. PIC18Fxxx 电路支持________个串行端口。
23. PIC18F458 片上程序 ROM 是________类型的。
24. PIC18F432 片上程序 ROM 是________类型的。
25. PIC18F452 片上程序 ROM 是________类型的。
26. PIC18F8772 片上程序 ROM 是________类型的。
27. 给出下面芯片的程序 ROM 与数据 RAM 的容量大小:
    (a) PIC18F2420　(b) PIC18F458　(c) PIC18F8772
28. 在 PIC18 系列里,如果你要在一个嵌入式产品上用上一百万片,哪种类型的内存成本最低?
29. PIC18F2420 和 PIC18F2220 的最主要区别是什么?
30. PIC18F458/4580 带有________字节的数据 EEPROM。

36

复习题答案

1.1 节

1. 正确
2. 基于微控制器的系统
3. (d)
4. (d)
5. 因为它只执行一种类型的工作。
6. 嵌入式系统是指应用和处理器都集成在同一片芯片上的系统。
7. 多源性意味着你可以从不只一个供应商处得到产品。另外,供应商之间的竞争有利于降低产品的价格。

1.2 节

1. 4 KB RAM 容量,2 MB 的片上 ROM 容量,还有大量的 I/O 引脚。
2. C 代表 OTP, 而 F 代表闪速 ROM。
3. PIC18F2420 带有 768 B 的 RAM,而 PIC18F4520 带有 1526 B 的 RAM。
4. (a) 16 KB (b) 32 KB
5. 8

第 2 章
PIC 体系结构与汇编语言编程

学习目标：

- ❑ 如何访问 PIC 微控制器的数据 RAM 文件寄存器
- ❑ 如何使用 WREG 和 MOVE 指令对数据进行操作
- ❑ 如何使用 PIC 微控制器的文件寄存器和存储器组进行 ADD 和 MOVE 等简单操作
- ❑ 状态寄存器的作用
- ❑ PIC 微处理器中数据 RAM 的空间分配
- ❑ PIC 微处理器的 SFR
- ❑ 如何编写简单的 PIC 汇编语言指令
- ❑ PIC 的数据类型和伪指令
- ❑ 如何使用 MPLAB 汇编和运行 PIC 程序
- ❑ PIC 上电时的引导过程
- ❑ 如何验证 PIC 程序 ROM 中的程序代码
- ❑ PIC 程序 ROM 的内存映射
- ❑ PIC 汇编语言指令的执行细节
- ❑ PIC 微控制器的 RISC 和哈佛结构
- ❑ 如何使用 MPLAB 仿真器验证 PIC 的寄存器和数据 RAM

CPU 使用大量的寄存器存储临时数据。要学习使用汇编语言编制程序，读者必须理解给定 CPU 的寄存器和体系结构，以及它们在数据处理中的作用。首先，2.1 节将介绍 PIC 的 WREG 寄存器。诸如 MOVE 和 ADD 这样简单的指令将用来作为例子，说明 PIC 单片机的最常用的寄存器的用途。2.2 节将分别讨论 PIC 的片上 RAM 的地址分配和 PIC18 的数据存储区。2.3 节将介绍如何编制程序访问数据存储区。2.4 节将介绍状态寄存器的标志位，以及算术运算指令对标志位的影响。2.5 节将描述 PIC 单片机的汇编语言指令、伪代码和数据类型。2.6 节将讨论汇编语言和机器语言编程，并定义有关术语，如助记符、操作码、运算域等。2.7 节将介绍为 PIC 编译和生成可运行程序的方法。2.8 节将探讨 PIC 程序的单步执行和程序计数器的作用。2.9 节将介绍 RISC 体系结构的优点。2.10 节将阐述如何使用 MPLAB 汇编和运行 PIC 程序。同时，也会讨论如何使用 MPLAB 仿真器查看寄存器和内存。

2.1 PIC 的 WREG 寄存器

PIC 微控制器带有很多用于算术运算和逻辑运算的寄存器，其中有一个 WREG 寄存器。

鉴于 PIC 中寄存器的数目众多，本节仅仅聚焦最为常用的 WREG 寄存器。GPR 和 SFR 将在下一节介绍。下面将描述 WREG 寄存器，并用简单指令 MOVE 和 ADD 来说明其功用。

2.1.1 WREG 寄存器

在 CPU 中，寄存器是用来存储临时信息的。这里所指的信息可以是待处理的字节数据，也可以是指向数据的地址指针。大多数 PIC 寄存器都是 8 位的。PIC 只有一种数据类型：8 位。8 位的寄存器如下面的方框图所示，按照从最高位（MSB）D7 到最低位（LSB）D 0 排列。任何长度超过 8 位的数据，在被处理前都会被截断成 8 位的字段。

D7	D6	D5	D4	D3	D2	D1	D0

8 位 WREG 寄存器是 PIC 微控制器中使用最广泛的寄存器。WREG 为工作寄存器，而且是唯一的一个工作寄存器。WREG 寄存器的作用相当于其他微处理器中的累加器，用于各种算术运算和逻辑运算指令。为了更好地理解 WREG 的使用，下面用两个简单的指令——MOVE 和 ADD——来举例说明。

40

2.1.2 MOVLW 指令

简单地说，MOVLW 指令是将 8 位数据传送至 WREG 寄存器。它的指令格式如下：

```
MOVLW K      ;move literal value K into WREG
```

K 是一个 8 位的数，范围是 0～255（十进制），或者是 00～FF（十六进制）。字母 L 是 literal 的缩写，代表字面值，必须是一个数值。换句话说，如果你在任何指令中看到“立即数”，就意味着需要在指令中正确的位置提供一个实际的数值。这类似于在其他微处理器中所见到的立即数。值得注意的是，在 MOVLW 指令中，字母 L 在前，字母 W 在后，也就是“把字面值传送至 WREG（目的地址）”。下面的指令将数值 25H（十六进制数）赋值给 WREG 寄存器。

```
MOVLW 25H    ;move value 25H into WREG (WREG = 25H)
```

下面的指令将数值 87H（十六进制）赋值给 WREG 寄存器。

```
MOVLW 87H    ;load 87H into WREG (WREG = 87H)
```

下面的指令将数值 15H（十六进制，折合十进制数为 21）赋值给 WREG 寄存器。

```
MOVLW 15H    ;load 15H into WREG (WREG = 15H)
```

2.1.3 ADDLW 指令

ADDLW 的指令格式如下：

```
ADDLW K      ;ADD literal value K to WREG
```

ADD 指令让 CPU 把字面值 K 加到寄存器 WREG，并把结果保存到 WREG 寄存器。值得注意的是，指令 AADLW 中的字母 L 在先，字母 W 紧随其后，即“把立即数加到 WREG（目的地址）”。要把两个数相加，如 25H 加 34H，可执行如下的指令：

```
MOVLW 25H    ;load 25H into WREG
ADDLW 34H    ;add value 34 to W(W = W + 34H)
```

执行完以上操作,得到 WREG=59H(25H+34H=59H)。

41

图 2-1 给出了字面值和 WREG 寄存器传送至 PIC ALU 单元的示意图。

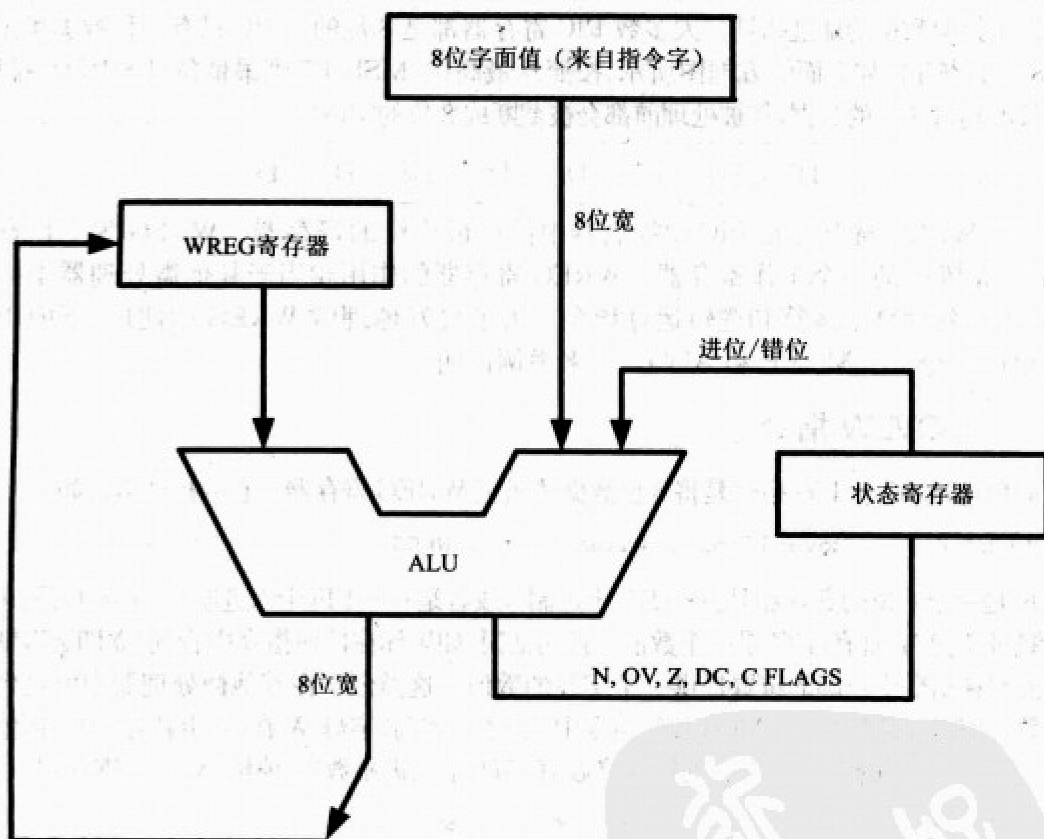


图 2-1 使用字面值的 PIC WREG 寄存器和 ALU

执行下面的程序,把 12H、16H、11H 和 43H 相加:

```
MOVLW 12H    ;load value 12H into WREG (WREG = 12H)
ADDLW 16H    ;add 16 to WREG (WREG = 28H)
ADDLW 11H    ;add 11 to WREG (WREG = 39H)
ADDLW 43H    ;add 43 to WREG (WREG = 7CH)
```

当编写将字面值传送至 WREG 寄存器的程序时,要注意以下几点。

(1)数值是直接送入 WREG 寄存器的。在其他微控制器中,通常需要在数值之前插入"$"或者"£"标志来识别立即数,但是对于 PIC WREG 寄存器并不需要。

(2)如果将数值(0~F)传送至 8 位寄存器(如 WREG),其余的二进制位将全被默认为 0。例如,执行指令"MOVLW 5H",结果是 WREG=05H,用二进制表示为 WREG=00000101。

(3)如果要把大于 255(FFH)的数值送入 WREG 寄存器,高位字节将会被截断,而且会在.err 文件中输出报警信息。

```
MOVLW 7F2H  ;ILLEGAL 7F2H > 8 bits (FFH), becomes F2H
MOVLW 456H  ;ILLEGAL 456H > FFH, becomes 56H
MOVLW 60A5H ;ILLEGAL but becomes A5H
```

42

2.1.4 复习题

1. 编写指令,将数值 34H 传送至 WREG 寄存器。
2. 编写指令,将数值 16H 和 CDH 相加,结果存放在 WREG 寄存器。
3. 判断正误:数值不可以直接传送至 WREG 寄存器。
4. 试问传送至 8 位寄存器的最大十六进制数是多少? 相应的十进制数又是多少呢?
5. PIC 中绝大部分寄存器都是________位的。

2.2 PIC 文件寄存器

除了 WREG 寄存器,PIC 微控制器还有很多其他的寄存器。这些寄存器都被称为数据存储器,以同程序存储器相区别。PIC 中的数据存储器是读/写存储器(静态 RAM)。在 PIC 微控制器文献中,数据存储器又被称为文件寄存器。本节将介绍 PIC 系列的文件寄存器空间分配,并用一些简单的指令(如 ADD 和 MOVE)来讨论它们的用处。

2.2.1 PIC 文件寄存器(数据 RAM)空间分配

文件寄存器是供 CPU 使用的读/写存储器,用来实现数据存储、数据暂存以及作为内部使用和实现功能的寄存器。正如之前提到的 WREG 寄存器,你可以对文件寄存器数据 RAM 里不同位置的数据进行算术运算与逻辑运算。PIC 微控制器的文件寄存器的容量大小因芯片而有所不同,从 32B 到几千字节不等。即使是同一系列,各芯片的文件寄存器的大小也是各不相同的。值得注意的是,和 WREG 一样,文件寄存器数据 RAM 也是 8 位宽的。PIC 的文件寄存器数据 RAM 分为两个类别:特殊功能寄存器和通用寄存器。通用寄存器部分又可以称为通用存储器(GP RAM)。下面将分别介绍这两类寄存器。

1. SFR(特殊功能寄存器)

SFR 是用于实现特殊功能的,例如 ALU 状态、定时器、串行通信、I/O 端口、ADC 模数转换等。由于 SFR 是用来控制微控制器或者外围设备的,所以每个 SFR 在设计之初就有明确的功能。PIC 的 SFR 是 8 位寄存器。文件寄存器为 SFR 所预留的地址数量取决于芯片的引脚数和它所支持的外部功能。在同一系列中,不同芯片所带有的 SFR 数量也不相同。有些芯片少至只有 7 个(如不带片上 ADC 模数转换器的 8 脚 PIC12C508),有些芯片多至上百个(如带有片上 ADC 模数转换器的 40PIC18F458)。举个例子,片上定时器数量越多,SFR 寄存器的数目也会越多。后面的章节将会介绍和用到更多的 SFR 寄存器。

43

2. GPR(通用寄存器或者 GPRAM)

GPR 是在用于数据存储和数据暂存的文件寄存器中的一组 RAM 地址空间。每个地址的数据宽度为 8 位,可以存储任何的 8 位数据。和之前提到的一样,GPR 的地址数量是文件寄存器根据芯片的引脚数目和它所支持的外部功能预留的。在 PIC 控制器中,没有分配给

SFR的空间都是可以用作GPR的。这意味着，假如一块PIC芯片有1000 B的文件寄存器，其中不到100 B已用作SFR，那么剩下的空间都分配给了通用寄存器。当使用汇编语言编程时，GPR容量越大意味着在寄存器管理上面临的困难越大。然而，今天的高性能微控制器的GPR已经超过了1KB，寄存器的管理工作可由C编译器完成。诚然，C编译器的使用是我们追求更大GPR的唯一原因，因为更大的GPR使得C编译器存储参数更容易、运行效率更高。表2-1比较了不同PIC芯片的文件寄存器。也可以参阅图2-2。

表2-1 PIC系列芯片的文件寄存器容量

	文件寄存器(单位为字节) =	SFR(单位为字节) +	GPR的可用空间(单位为字节)
PIC12F508	32	7	25
PIC16F84	80	12	68
PIC18F1220	512	256	256
PIC18F452	1792	256	1536
PIC18F2220	768	256	512
PIC18F458	1792	256	1536
PIC18F8722	4096	158	3938

摘自 http://www.microchip.com。

44

Microchip的网站提供了数据RAM容量，它和GPR容量一样大。

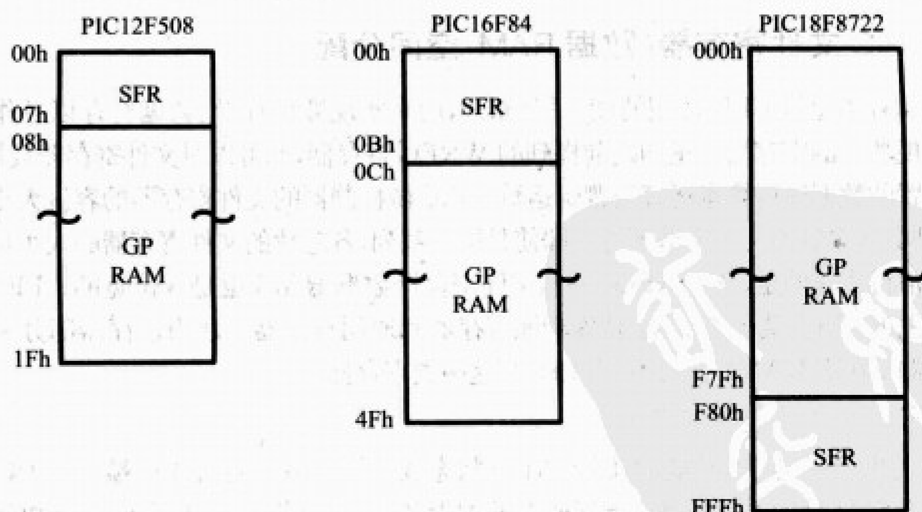


图2-2 PIC12、PIC16、PIC18的文件寄存器

2.2.2 PIC芯片中的GP RAM和EEPROM比较

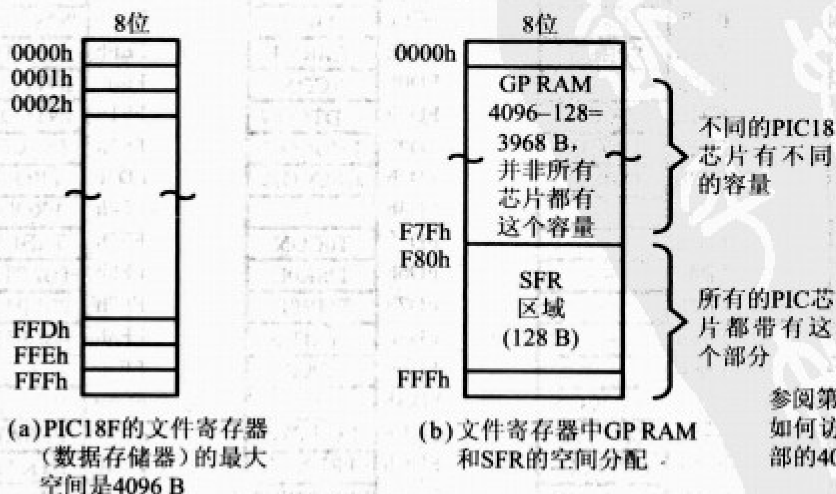
值得注意的是，在Microchip公司网站的芯片信息网页上，有两栏不同的RAM信息，其中一栏是通用寄存器(GP RAM)的大小；另一栏是EEPROM的大小。一定不要将GP RAM(组成了文件寄存器的大部分)和EEPROM数据存储器混为一谈。GPR是供CPU存储内部数据的，而EEPROM是芯片的外加存储器。换而言之，许多PIC芯片可以不带EEPROM数据存储器，但微控制器不能不带文件寄存器。第14章将会介绍PIC芯片的EEPROM存储器。

2.2.3 PIC18 的文件寄存器与访问存储区

PIC18 系列的文件寄存器最大容量可达 4096(4K)B。4096B 意味着文件寄存器的地址范围为 000H～FFFH。PIC18 的文件寄存器被分成大小为 256 B 的存储区。因此，最多可以有 16 个存储区(16×256=4096)。虽然并非所有的 PIC18 芯片都有这么多存储区，但是每个芯片至少会有一个存储区用作文件存储器。该存储区被称为访问存储区，而且是芯片上电时的默认存储区。对于如何使用 PIC 系列的文件寄存器，为简化起见，本章聚焦于 PIC18 系列中每个芯片均带有的这个默认存储区。也可以在 Microchip 公司的网站上查找到其他 PIC 系列(如 PIC12 或者 PIC16)的文件寄存器资料。尽管本书重点是介绍带有大容量文件寄存器的 PIC18 系列，但是其讨论的基本思想同样可以用于 PIC12 系列和 PIC16 系列。

观察如图 2-3 所示的 PIC18 的访问存储区可知，256B 的访问存储区被分成了大小相等的两个区域。这两个 128 B 的区域分别对应着 GPR 和 SFR。位于 00H～7FH 的 128 B 是 GPR，用作读/写存储或者数据暂存。该 128 B 的 RAM 常用来存储 PIC18 程序和 C 编译器的数据和参数。GPR 的每个地址都是可以直接寻址的。在以后的章节中你会发现，这些地址可用来存储经由 I/O 端口或串行端口传送至 CPU 的数据。第 3 章还使用它们来定义反映延时的计算器。其余的 128 B 访问存储区是用作 SFR 的，地址范围是 F80H～FFFH，如图 2-4 所示。或许有人会问：为什么访问存储区中 SFR 和 GPR 的地址不是空间连续的？原因是，当 PIC18 文件存储器需要存储大量的数据时，RAM 空间 080H～F7FH 就可以用于 GPR。大于 256 B 的文件存储器必须进行存储区切换。当 PIC18 芯片有多于最小访问存储区时，采用存储区切换的方法可以访问文件寄存器中的所有存储区。第 6 章在讨论组切换时，将会更详细地讨论文件寄存器容量大于 256 B 的 PIC18 芯片。

注意，I/O 端口 SFR、PROTA、PORTB、PORTC、PORTD 和相关的寄存器都是 PIC 中广泛使用的 SFR。关于 SFR 的其他信息，请参阅第 4 章。



(a) PIC18F的文件寄存器（数据存储器）的最大空间是4096 B

(b) 文件寄存器中GP RAM和SFR的空间分配

图 2-3 PIC18 系列的文件寄存器

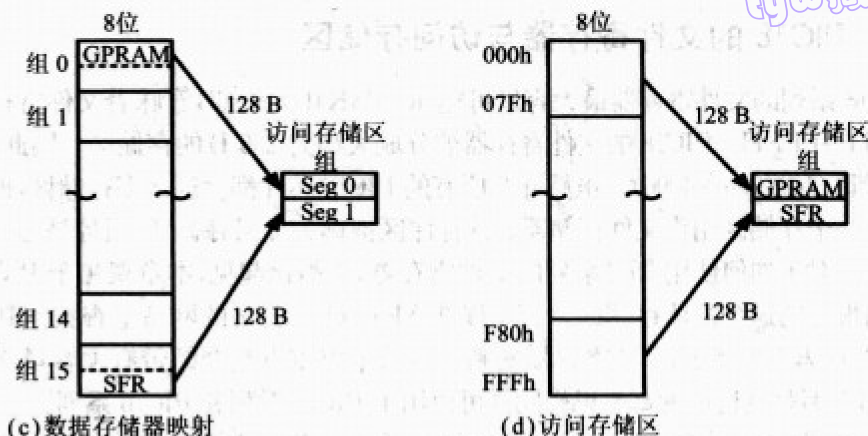


图 2-3　(续)

45
~
46

地址	寄存器	地址	寄存器	地址	寄存器	地址	寄存器
F80h	PORTA	FA0h	PIE2	FC0h	—	FE0h	BSR
F81h	PORTB	FA1h	PIR2	FC1h	ADCON1	FE1h	FSR1L
F82h	PORTC	FA2h	IPR2	FC2h	ADCON0	FE2h	FSR1H
F83h	PORTD	FA3h	—	FC3h	ADRESL	FE3h	PLUSW1 *
F84h	PORTE	FA4h	—	FC4h	ADRESH	FE4h	PREINC1 *
F85h	—	FA5h	—	FC5h	SSPCON2	FE5h	POSTDEC1 *
F86h	—	FA6h	—	FC6h	SSPCON1	FE6h	POSTINC1 *
F87h	—	FA7h	—	FC7h	SSPSTAT	FE7h	INDF1 *
F88h	—	FA8h	—	FC8h	SSPADD	FE8h	WREG
F89h	LATA	FA9h	—	FC9h	SSPBUF	FE9h	FSR0L
F8Ah	LATB	FAAh	—	FCAh	T2CON	FEAh	FSR0H
F8Bh	LATC	FABh	RCSTA	FCBh	PR2	FEBh	PLUSW0 *
F8Ch	LATD	FACh	TXSTA	FCCh	TMR2	FECh	PREINC0 *
F8Dh	LATE	FADh	TXREG	FCDh	T1CON	FEDh	POSTDEC0 *
F8Eh	—	FAEh	RCREG	FCEh	TMR1L	FEEh	POSTINC0 *
F8Fh	—	FAFh	SPBRG	FCFh	TMR1H	FEFh	INDF0 *
F90h	—	FB0h	—	FD0h	RCON	FF0h	INTCON3
F91h	—	FB1h	T3CON	FD1h	WDTCON	FF1h	INTCON2
F92h	TRISA	FB2h	TMR3L	FD2h	LVDCON	FF2h	INTCON
F93h	TRISB	FB3h	TMR3H	FD3h	OSCCON	FD3h	PRODL
F94h	TRISC	FB4h	—	FD4h	—	FF4h	PRODH
F95h	TRISD	FB5h	—	FD5h	T0CON	FF5h	TABLAT
F96h	TRISE	FB6h	—	FD6h	TMR0L	FF6h	TBLPTRL
F97h	—	FB7h	—	FD7h	TMR0H	FF7h	TBLPTRH
F98h	—	FB8h	—	FD8h	STATUS	FF8h	TBLPTRU
F99h	—	FB9h	—	FD9h	FSR2L	FF9h	PCL
F9Ah	—	FBAh	CCP2CON	FDAh	FSR2H	FFAh	PCLATH
F9Bh	—	FBBh	CCPR2L	FDBh	PLUSW2 *	FFBh	PCLATU
F9Ch	—	FBCh	CCPR2H	FDCh	PRENC2 *	FFCh	STKPTR
F9Dh	PIE1	FBDh	CCP1CON	FDDh	POSTDEC2 *	FFDh	TOSL
F9Eh	PIR1	FBEh	CCPR1L	FDEh	POSTINC2 *	FFEh	TOSH
F9Fh	IPR1	FBFh	CCPR1H	FDFh	INDF2 *	FFFh	TOSU

* 这些都不是物理寄存器。

图 2-4　PIC18 系列的 SFR

2.2.4 复习题

1. 判断对错：PIC 中的数据空间是 SRAM 型存储器，而程序(代码)空间是 ROM 型的。
2. 通用 RAM 和 SFR 统称为________。
3. 判断对错：文件寄存器越大，管理它们就越困难。
4. 判断对错：分配给 SFR 的文件寄存器空间越大，GP RAM 的可用空间就越小。
5. PIC 系列的 SFR 是________位的。
6. PIC 的文件寄存器空间被分成________字节的存储区。
7. PIC 的文件寄存器空间最大可以是________字节。

47

2.3 默认访问存储区的指令操作

到现在为止，本书所介绍的指令都是关于字面值(常数)K 和 WREG 寄存器的。它们将 WREG 寄存器作为目的地址，在 2.1 节就介绍了使用 MOVLM 和 ADDLW 指令的简单例子。PIC 允许 ALU 和其他运算直接寻址文件寄存器的其他地址空间。本节将介绍使用文件寄存器不同地址的指令。对于读者掌握 PIC 汇编语言来说，本节是全书最重要的部分内容之一。

2.3.1 MOVWF 指令

正如上一节所提到的，文件寄存器的访问存储区在 PIC18 上电时是默认的存储区。必须强调术语文件寄存器的原因是，对它进行操作的指令助记符中含有字母 F。在 MOVWF 这样的指令中，字母 F 代表文件寄存器中的一个地址，而字母 W 代表 WREG 寄存器。指令 MOVWF 的含义是，让 CPU 将源寄存器 WREG 中的内容传送(实际上是复制)至文件寄存器(F)的目的地址。执行完该指令后，文件寄存器的相应地址上的数据和 WREG 寄存器中的值是相同的。该文件寄存器的地址可以是某个 SFR，也可以是通用寄存器的地址。例如，指令 MOVWF PORTA 就是将 WREG 寄存器中的值传送到名为 PORTA 的 SFR 寄存器。下面的程序首先把数值 55H 赋给 WREG 寄存器，然后再将该值依次传送至不同的 SFR(*B* 端口、*C* 端口、*D* 端口)：

```
MOVLW 55H         ;WREG = 55H
MOVWF PORTB       ;copy WREG to Port B (Port B = 55H)
MOVWF PORTC       ;copy WREG to Port C (Port C = 55H)
MOVWF PORTD       ;copy WREG to Port D (Port D = 55H)
```

PORTB、PORTC、PORTD 都是文件寄存器中 SFR 的组成部分，如图 2-4 所示。第 4 章将介绍它们可以连接 PIC 微控制器的 I/O 引脚。还可以将 WREG 寄存器中的内容移动(复制)到文件寄存器的通用寄存器(RAM)中的任何位置。下面的程序是将 99H 传送到文件寄存器中 GPR 区域的地址 0～4：

```
MOVLW 99H     ;WREG = 99H
MOVWF 0H      ;move (copy) WREG contents to location 0h
MOVWF 1H      ;move (copy) WREG contents to location 1h
MOVWF 2H
MOVWF 3H
MOVWF 4H
```

地址	数值
000	99
001	99
002	99
003	99
004	99

48

上表说明了代码执行后地址 0~4 的内容。

例 2-1 执行下面的程序，确定文件寄存器各地址的内容。

```
MOVLW 99H        ;load WREG with value 99H
MOVWF 12H
MOVLW 85H        ;load WREG with value 85H
MOVWF 13H
MOVLW 3FH        ;load WREG with value 3FH
MOVWF 14H
MOVLW 63H        ;load WREG with value 63H
MOVWF 15H
MOVLW 12H        ;load WREG with value 12H
MOVWF 16H
```

解：

当执行完 MOVWF 12H 时，文件寄存器中地址 12H 的值为 99H；
当执行完 MOVWF 13H 时，文件寄存器中地址 13H 的值为 85H；
当执行完 MOVWF 14H 时，文件寄存器中地址 14H 的值为 3FH；
当执行完 MOVWF 15H 时，文件寄存器中地址 15H 的值为 63H；
等等，结果如右表所示。

地址	数值
012	99
013	85
014	3F
015	63
016	12

注意，字面值（立即数）不能被直接传送至 PIC18 的通用寄存器 RAM 地址。它们必须通过 WREG 寄存器进行传送。

2.3.2 关于 WREG 和访问存储区的更多指令

实际上，还有一组逻辑运算和算术运算指令是涉及 WREG 寄存器和文件寄存器地址的。指令 ADDWF 就是其中之一。ADDWF 的功能是将 WREG 寄存器和文件寄存器地址中的内容相加。该文件寄存器地址是一个 SFR 寄存器或者一个通用寄存器。存放结果的目的地址是 WREG 寄存器或者文件寄存器。下面的指令格式将表明目的地址：

```
ADDWF fileReg, D
```

其中，fileReg 表示文件寄存器地址，D 表示目的位，可以是 0 也可以是 1。当 D=0 时，说明结果存放在 WREG 寄存器；当 D=1 时，说明结果将存放在文件寄存器。

下面的程序先把 22H 传送到通用寄存器的地址 5、6 和 7，然后把它们相加，再把结果送入 WREG 寄存器：

```
MOVLW 22H    ;WREG = 22H
MOVWF 5H     ;move(copy) WREG contents to location 5H
MOVWF 6H     ;move(copy) WREG contents to location 6H
MOVWF 7H     ;move(copy) WREG contents to location 7H
ADDWF 5H, 0 ;add W and loc 5, result in WREG (W = 44H)
ADDWF 6H, 0 ;add W and loc 6, result in WREG (W = 66H)
ADDWF 7H, 0 ;add W and loc 7, result in WREG (W = 88H)
```

地址	数值
005	22
006	22
007	22

执行完 MOVWF 7H 后 GPR 的状态为：WREG = 22H。

地址	数值
005	22
006	22
007	22

执行完 ADDWF 7H，0 后 GPR 的状态为：WREG = 88H。

下面的程序完成同样的功能，但结果存放在文件寄存器的地址 7：

```
MOVLW 22H    ;WREG = 22H
MOVWF 5H     ;move (copy) WREG contents to location 5H
MOVWF 6H     ;move (copy) WREG contents to location 6H
MOVWF 7H     ;move (copy) WREG contents to location 7H
ADDWF 5, 0   ;add W and loc 5, result in WREG (W = 44H)
ADDWF 6, 0   ;add W and loc 6, result in WREG (W = 66H)
ADDWF 7, 1   ;add W and loc 7, result in location 7H
             ;now location 7 has 88H and WREG = 66H
```

地址	数值
005	22
006	22
007	22

执行完 MOVWF 7H 后 GPR 的状态为：WREG=22H。

地址	数值
005	22
006	22
007	88

执行完 ADDWF 7H，1 后 GPR 的状态为：WREG = 66H。

50

尽管已经引入了助记符 D，但是为了避免混淆，PIC 汇编器允许使用 W 和 F(代替 0 或者 1)来定义目的地址。请看下面两种指令格式：

```
ADDWF fileReg, w  ;add WREG and fileReg. WREG = the result
ADDWF fileReg, f  ;add WREG and fileReg
                  ;fileReg = the result
```

这种指令格式更为简明，而且能避免混淆目的地址。下面采用新的指令格式重新编写上面的程序为：

```
MOVLW 22H    ;WREG = 22H
MOVWF 5H     ;move (copy) WREG contents to location 5H
MOVWF 6H     ;move (copy) WREG contents to location 6H
MOVWF 7H     ;move (copy) WREG contents to location 7H
ADDWF 5H,W   ;add W and loc 5, result in WREG (W = 44H)
ADDWF 6H,W   ;add W and loc 6, result in WREG (W = 66H)
ADDWF 7H,F   ;add W and loc 7, result in location 7
             ;now location 7 has 88H and WREG = 66H
```

上述的概念非常重要，并且要理解透彻，因为 PIC18 中还有很多指令都是这种格式的。下面比较例 2-2 和例 2-3。

例 2-2 执行下面的程序，分别确定文件寄存器中地址 12H 和 WREG 寄存器的内容。

```
MOVLW 0          ;move 0 WREG to clear it (WREG = 0)
MOVWF 12H        ;move WREG to location 12 to clear it
```

```
MOVLW 22H          ;load WREG with value 22H
ADDWF 12H, F       ;add WREG to loc 12H, loc 12 = sum
ADDWF 12H, F       ;add WREG to loc 12H, loc 12 = sum
ADDWF 12H, F       ;add WREG to loc 12H, loc 12 = sum
ADDWF 12H, F       ;add WREG to loc 12H, loc 12 = sum
```

解：

该程序首先把 WREG 寄存器和文件寄存器的地址 12H 清零，然后把 WREG 赋值为 22H，再把 WREG 寄存器和地址 12H 的内容相加，并把结果放入地址 12H。共重复 4 次。最后，通用寄存器地址 12H 的内容为 88H(4×22H=88H)，WREG 值为 22H。

执行每一条指令"ADDWF 12,F "后的情况如下表所示。

地址	数值
011	
012	22
013	

WREG = 22H

地址	数值
011	
012	44
013	

WREG = 22H

地址	数值
011	
012	66
013	

WREG = 22H

地址	数值
011	
012	88
013	

WREG = 22H

51

例 2-3 重写上面的例子，把文件寄存器和 WREG 寄存器的内容相加结果放入 WREG 寄存器。

```
MOVLW 0            ;move 0 WREG to clear it (WREG = 0)
MOVWF 12H          ;move WREG to location 12 to clear it
MOVLW 22H          ;load WREG with value 22H
ADDWF 12H, W       ;add WREG and loc 12H, WREG = sum
ADDWF 12H, W       ;add WREG and loc 12H, WREG = sum
ADDWF 12H, W       ;add WREG and loc 12H, WREG = sum
ADDWF 12H, W       ;add WREG and loc 12H, WREG = sum
```

解：

程序每次把 WREG 寄存器和地址 12H 的内容相加，然后把结果放入 WREG。最后地址 12H 的内容为 22H，而 WREG 寄存器的内容为 88H(4×22H=88H)。

执行每一条指令"ADDWF 12,W"后的情况如下表所示。

地址	数值
011	
012	22
013	

WREG = 22H

地址	数值
011	
012	22
013	

WREG = 44H

地址	数值
011	
012	22
013	

WREG = 66H

地址	数值
011	
012	22
013	

WREG = 88H

52

现在开始讨论表 2-2 和表 2-3 所示的指令。表 2-2 中的指令适用于 WREG 和文件寄存器地址，并且可以选择把结果放在 WREG 寄存器或者文件寄存器地址中。但表 2-3 中的指令只能对文件寄存器进行操作，然后选择把结果放在 WREG 寄存器还是文件寄存器中。

表 2-2 能同时对 WREG 和 fileReg 操作的 ALU 指令

指　　令		
ADDWF	fileReg, d	将 WREG 的内容和 fileReg 的内容相加
ADDWFC	fileReg, d	将带进位的 WREG 的内容和 fileReg 的内容相加
ANDWF	fileReg, d	将 WREG 的内容和 fileReg 的内容进行与操作
IORWF	fileReg, d	将 WREG 的内容和 fileReg 的内容进行或操作
SUBFWB	fileReg, d	将带借位的 WREG 的内容减去 fileReg 的内容
SUBWF	fileReg, d	将 fileReg 的内容减去 WREG 的内容
SUBWFB	fileReg, d	将带借位的 fileReg 的内容减去 WREG 的内容
XORWF	fileReg, d	将 WREG 的内容和 fileReg 的内容进行异或操作

注意：位 d 用来决定运算的目的地址。当 d＝w 时，结果放入 WREG (d＝0)。当 d＝F 时，结果放入 fileReg (d＝1)。默认值是 F，即 `ADDWF myfile` 相当于 `ADDWF myfile, F`。

有关表 2-2 所列指令的例子，请参阅第 5 章。

表 2-3 采用 fileReg 或 WREG 为目的地址的文件寄存器指令

指　　令		
COMF	fileReg, d	fileReg 取反
DECF	fileReg, d	fileReg 自减 1
DECFSZ	fileReg, d	若 fileReg 不为 0，减 1
DECFSNZ	fileReg, d	若 fileReg 为 0，减 1
INCF	fileReg, d	fileReg 自加 1
INCFSZ	fileReg, d	若 fileReg 不为 0，加 1
INCSNZ	fileReg, d	若 fileReg 为 0，加 1
MOVF	fileReg, d	传送 fileReg
NEGF	fileReg, d	fileReg 取相反数
RLCF	fileReg, d	带进位循环左移
RLNCF	fileReg, d	循环左移(不带进位)
RRCF	fileReg, d	带进位循环右移
RRNCF	fileReg, d	循环左移(不带进位)
SWAPF	fileReg, d	fileReg 中半字节交换
BTG	fileReg, d	位跳变 fileReg

注意：位 d 用来决定运算的目的地址。当 d＝w 时，结果放入 WREG (d＝0)。当 d＝F 时，结果放入 fileReg (d＝1)。默认值是 F，即 `DECF myfile` 相当于 `DECF myfile, F`。

关于如何使用表 2-3 中的指令的内容，将在第 3 章～第 6 章中介绍。

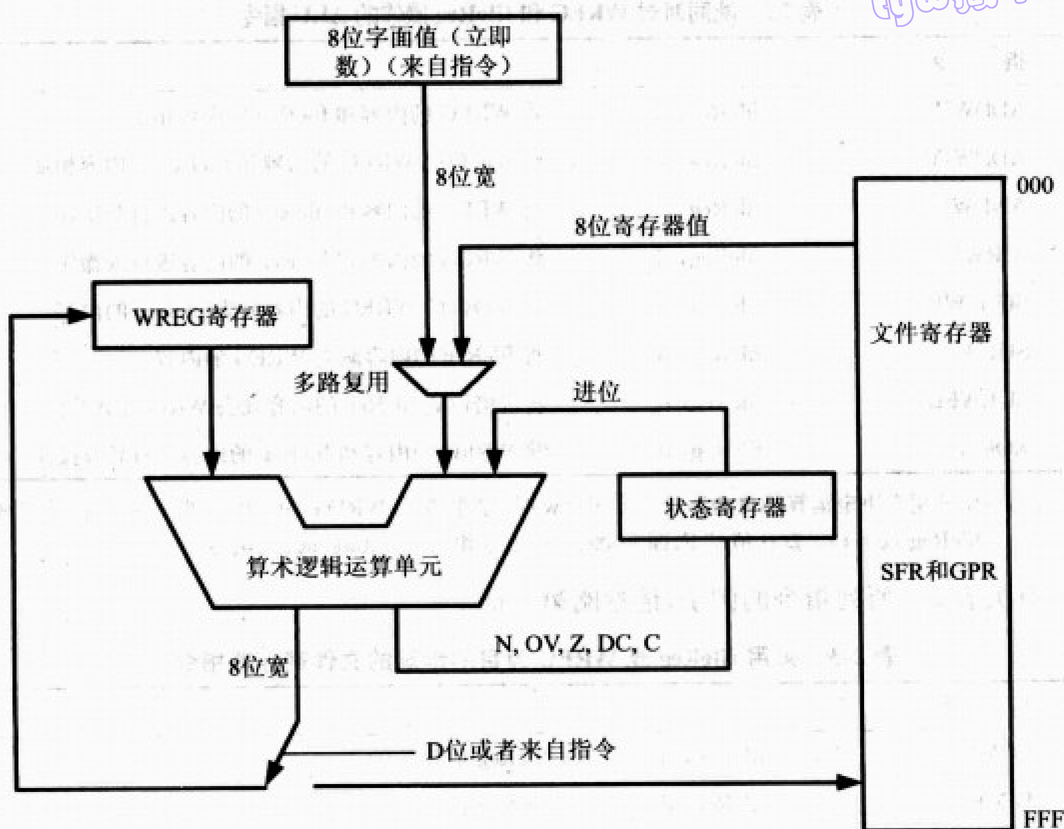


图 2-5 PIC18 的 WREG、fileReg 和 ALU

2.3.3 COMF 指令

指令 COMF fileReg, d将 fileReg 的值取反，然后放在 WREG 寄存器或 fileReg 中。这是一个“读—修改—写”的例子，更多的例子将在以后的章节中见到。下面的程序把数值 55H 赋给 WREG 寄存器，然后送入 SFR 的 PORTB，再把 PORTB 的值取反，最后得到 AAH。于是，数值 01010101(55H)取反后变为 10101010(AAH)。

```
MOVLW  55H         ;WREG = 55h
MOVWF  PORTB       ;Move WREG to Port B SFR (PB = 55h)
COMF   PORTB, F    ;complement Port B (PB = AAh)
```

请看下面的例 2-4。

例 2-4 编写一个简单的程序，跳变 PORTB 的 SFR，并且让它永远持续下去。

```
      MOVLW  55H         ;WREG = 55h
      MOVWF  PORTB       ;move WREG to Port B SFR (PB = 55h)
B1    COMF   PORTB, F    ;complement Port B and place it in Port B
      GOTO   B1          ;repeat forever (See Chapter 3 for GOTO)
```

2.3.4 DECF 指令

指令 DECF fileReg, d 让 fileReg 减 1,然后把结果放入 fileReg 或者 WREG 寄存器。下面的程序把数值 3 送入 fileReg 地址 0x20,然后将地址 0x20 的值自减 1,所得结果再送回至 fileReg。

```
MOVLW  3            ;WREG = 3
MOVWF  20H          ;move WREG to loc 20H (loc 20H = 3)
DECF   0x20, F      ;loc 20H has 2
DECF   0x20, F      ;loc 20H has 1
DECF   0x20, F      ;loc 20H has 0 and WREG = 3
```

现在,比较上面的代码和下面的程序。

```
MOVLW  3            ;WREG = 3
MOVWF  20H          ;move WREG to loc 20H (loc 20H = 3)
DECF   0x20, W      ;loc 20H has 3 (WREG = 2)
DECF   0x20, W      ;loc 20H has 3 (WREG = 2)
DECF   0x20, X      ;loc 20H has 3 (WREG = 2)
```

上面的概念将会用在下一章所介绍的循环操作中。

54

2.3.5 MOVF 指令

助记符 MOVF 其实是执行 MOVFW 的功能,它的指令格式如下:

```
MOVF        fileReg, D
```

如果 D=0,fileReg 的内容被复制到 WREG 寄存器中;如果 D=1,那么 fileReg 的内容就被送回到 fileReg 自身中。尽管通常情况下使用 MOVF 指令把来自 I/O 引脚的数据送入 WREG 寄存器,但有时候也会使用 MOVF 指令将 fileReg 的内容传送给它自身,以测试 fileReg的内容。下面考查指令 MOVWF 和 MOVF 的区别。在上文中,使用 MOVWF 可以很容易地将数据写入 SFR(如端口 *B*)。而且,采用 MOVWF 指令将固定值(字面值)加载到文件寄存器的 RAM 空间也是惯常的做法,因为没有其他方法能直接对文件寄存器加载数据。相比较而言,MOVF 指令主要用来把来自 I/O 端口(如 *B* 端口)的数据送入到 CPU。此外,也可以用 MOVF 指令把来自 SFR 或者 GP RAM 中任何地址的数据传送至 WREG 寄存器,以执行算术或者逻辑运算。请看下面的例 2-5 和例 2-6。注意,唯一的一次用到 MOVF fileReg, F 指令将数据从 fileReg 复制到它自身,是因为改变状态寄存器标志位的需要。有关状态寄存器标志位的内容将在下一节介绍,第 3 章还会讨论如何使用状态寄存器标志位。

例 2-5 编写程序,把来自 PORTB 的 SFR 里的数据持续不断地送入的 PORTC SFR 中。

解:

```
AGAIN MOVF  PORTB, W    ;bring data from PortB into WREG
      MOVWF PORTC       ;send it to Port C
      GOTO  AGAIN       ;keep doing it forever
```

例 2-5 使用 GOTO 指令来不断重复传送操作。第 3 章将会介绍循环操作,而更多 I/O 端口的内容将在第 4 章中介绍。

例 2-6　编写程序，从 *B* 端口的 SFR 中读入数据，加上立即数 5 后将所得结果送入 *C* 端口的 SFR。

解：

```
MOVF   PORTB,W       ;bring data from Port B into WREG
ADDLW  05H           ;add 5 to WREG
MOVWF  PORTC         ;copy WREG to Port C
```

2.3.6　MOVFF 指令

MOVFF 指令是将数据从 fileReg 的一个地址传送到 fileReg 的另一个地址。作为源地址和目的地址的 fileReg 的地址可以是 PIC18 数据 RAM 里 4096 个地址中的任何一个。MOVFF 允许数据在 4KB RAM 空间内任意传送而无需经由 WREG 寄存器，如图 2-6 所示。接下来比较例 2-5 和例 2-7。

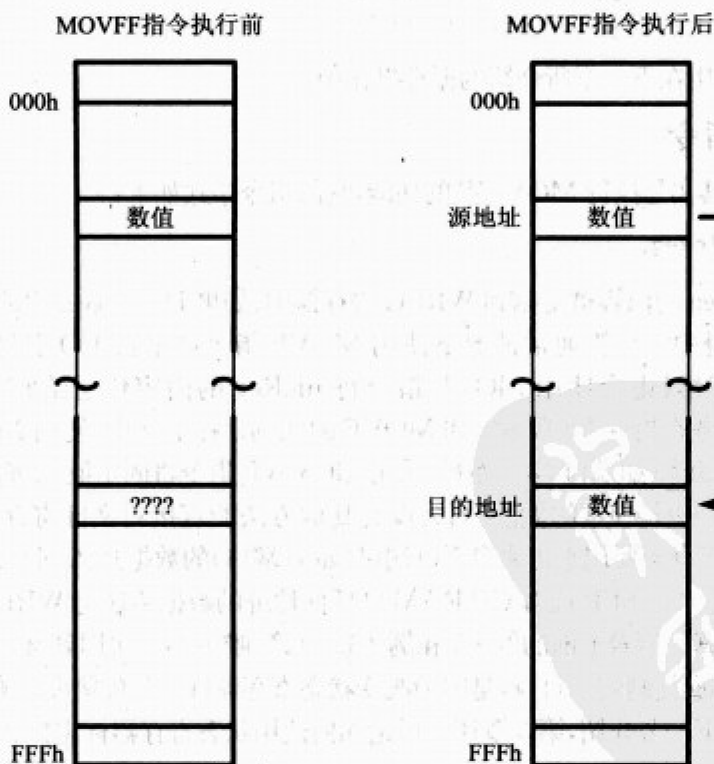


图 2-6　在 fileReg 空间里直接传送数据

例 2-7　编写程序，使用 MOVFF 指令把 *B* 端口的 SFR 中的数据不断送入 *C* 端口的 SFR 中。并同例 2-5 比较，解释它们的区别。

解：

```
AGAIN MOVFF PORTB, PORTC        ;copy data from Port B to Port C
      GOTO AGAIN                ;keep doing it forever
```

在例2-5中，有

```
AGAIN MOVF  PORTB, W        ;bring data from Port B into WREG
      MOVWF PORTC           ;send it to Port C
      GOTO  AGAIN           ;keep doing it forever
```

使用MOVFF指令可以简单地把数据从一个地址复制到另一个地址。但是，若用WREG寄存器，则还可以在传送数据之前进行算术运算或逻辑运算。

56

2.3.7 复习题

1. 判断对错：访问存储器是在GPR和SFR之间平均分配的256 B空间。
2. 编写指令，求16H与CDH的和。将所得的结果放入文件寄存器的地址0。
3. 判断对错：数值不能直接传送至通用寄存器。
4. 文件寄存器里能传送至某一地址的最大十六进制数是多少？相对应的十进制数是多少？
5. 指令 ADDWF PORTB, W 的结果放在________。

2.4 PIC 状态寄存器

和所有的微处理器一样，PIC有一个标志寄存器来记录算术运算条件(如进位)。PIC的标志寄存器又被称为**状态寄存器**。本节将讨论状态寄存器的每个标志位的功能，并给出改变这些标志位的例子。第3章和第5章将会讨论各个标志位的使用。

2.4.1 PIC18 状态寄存器

PIC18的状态寄存器是一个8位寄存器，有时又被称作**标志寄存器**。虽然状态寄存器共有8个标志位，但PIC18只用到了其中的5位。未使用的3位是无效位，所读状态为0。所使用的5个标志位又被称作**条件状态标志位**，意指执行指令后PIC18的状态。这5个标志位分别是C(进位标志)、DC(数字进位标志)、Z(零标志)、OV(溢出标志)、N(负数标志)。状态寄存器的标志位分配如图2-7所示。每一个条件标志位都可用于执行一个条件转移(分支)操作。更多信息请参阅第3章。

下面首先简要介绍状态寄存器的各个标志位，然后讨论指令对状态寄存器的影响。

1. C，进位标志位

当D7位产生进位时，该标志位将被置1。执行8位的加法或者减法运算都可能影响该标志位。第5章将会介绍进位标志位的用法。



C-进位标志位
DC-数字进位标志位
Z-零标志位
OV-溢出标志位
N-负数标志位
X-D5、D6、D7未定义，保留

图2-7 状态寄存器的标志位

57

2. DC,数字进位标志位

在ADD或SUB操作过程中,如果有自D3向D4的进位,该标志位将被置1,否则清零。当执行BCD码运算时会用到该标志位。有些微处理器称它为AC标志位(辅助进位/借位标志位)。更多的介绍请参阅第5章。

3. Z,零标志位

零标志位反映算术运算或者逻辑运算的结果。若结果为零,则Z=1;若结果不为零,则Z=0。关于循环程序中如何使用零标志位,请参阅第3章。

4. OV,溢出标志位

当且仅当有符号数的运算结果太大、导致高位溢出到符号位时,该标志位会被置1。通常,进位标志位是用于检测无符号数算术运算的错误的,而溢出标志位是用来检测有符号数算术运算的错误的。OV和N标志位都是用于有符号数的算术运算操作的,这将在第5章中介绍。

5. N,负数标志位

有符号数的二进制数表示用D7位作为符号位。负数标志位反映算术运算的结果。如果D7位为0,那么N=0,结果是正数。如果D7位是1,那么N=1,结果为负数。N和OV标志位都是用于有符号数的算术操作的,这将在第5章中介绍。

2.4.2 ADDLW指令和状态寄存器

接下来将讨论ADDLW指令对标志位C、DC和Z的影响,通过例子的形式来弄清它们的含义。尽管所有标志位(C、Z、DC、OV和N)都会受到ADDLW指令的影响,但现在只集中讨论C、DC和Z标志位。鉴于其他的标志位只涉及有符号数的运算,因此将安排在第5章讨论。通过例2-8到例2-10,将会弄清ADD指令是如何影响这3个标志位的。

2.4.3 并非所有指令都会影响标志位

有些指令对C、DC、Z、OV和N都会产生影响(如ADDWL)。但是,有些指令对标志位毫无影响。例如,数据传送指令就是其中之一(MOVF除外)。有些指令只会影响Z或N;逻辑指令就是这种类型,如ANDWL。

58

例2-8 在下面的指令中,执行38H加2FH后,试确定标志位C、DC和Z的状态。

```
MOVLW 38H
ADDLW 2FH          ;add 2FH to WREG
```

解:

38H	0011	1000	
+2FH	0010	1111	
67H	0110	0111	WREG=67H

C=0,因为D7位没有产生进位。

DC=1,因为D3向D4进位了。

Z=0,因为执行加法运算后,WREG寄存器的内容不为0。

例 2-9 在下面的指令中，执行 9CH 加 64H 后，试确定标志位 C、DC 和 Z 的状态。

```
MOVLW  9CH
ADDLW  64H          ;add 64H to WREG
```

解：

```
      9CH     1001   1100
   +  64H     0110   0100
     100H     0000   0000     WREG=00
```

C=1，因为 D7 位有进位产生。

DC=1，因为 D3 向 D4 进位了。

Z=1，因为执行加法运算后，WREG 的内容为 0。

例 2-10 在下面的指令中，执行 88H 加 93H 后，试确定标志位 C、DC 和 Z 的状态。

```
MOVLW  88H
ADDLW  93H          ;add 93H to WREG
```

解：

```
      88H     1000   1000
   +  93H     1001   0011
     11BH     0001   1011     WREG=1BH
```

C=1，因为 D7 位有进位产生。

DC=0，因为 D3 没有向 D4 进位。

Z=0，因为执行加法运算后，WREG 的内容为 0。

表 2-4 列出了运算指令以及受它们影响的标志位。附录 A 给出了完整的指令清单和受它们影响的标志位。

表 2-4 影响标志位的指令

指令	C	DC	Z	OV	N
ADDLW	X	X	X	X	X
ADDWF	X	X	X	X	X
ADDWFC	X	X	X	X	X
ANDLW			X		X
ANDWF			X		X
CLRF			X		
COMF			X		X
DAW	X				
DECF	X	X	X	X	X
INCF	X	X	X	X	X
IORLW			X		X
IORWF			X		X

（续）

指　　令	C	DC	Z	OV	N
MOVF			X		
NEGF	X	X	X	X	X
RLCF	X		X		X
RLNCF			X		X
RRCF	X		X		X
RRNCF			X		X
SUBFWB	X	X	X	X	X
SUBLW	X	X	X	X	X
SUBWF	X	X	X	X	X
SUBWFB	X	X	X	X	X
XORLW			X		X
XORWF			X		X

注意：X 可以是 0 或 1。

关于这些指令的用法，请参阅第 5 章。

2.4.4　标志位和判决

因为状态标志位也称为条件标志位，所以有一部分指令是根据标志位的状态来执行条件跳转（分支）指令的。表 2-5 给出了这些指令的清单。条件分支指令及其用法将在第 3 章中讨论。

表 2-5　使用标志位的 PIC18 分支（跳转）指令

指　　令	操　　作	指　　令	操　　作
BC	若 C=1 则跳转	BN	若 N=1 则跳转
BNC	若 C≠0 则跳转	BNN	若 N≠0 则跳转
BZ	若 Z=1 则跳转	BOV	若 OV=1 则跳转
BNZ	若 Z≠0 则跳转	BNOV	若 OV≠0 则跳转

2.4.5　复习题

1. PIC 的标志寄存器称为________。
2. PIC 标志寄存器的大小是多少？
3. 状态寄存器的哪几位是未使用的？
4. 确定执行下面指令后 C、Z、DC 标志位的状态：

```
MOVLW  9FH
ADDLW  61H
```

5. 确定执行下面指令后 C、Z、DC 标志位的状态：

```
MOVLW  82H
ADDLW  22H
```

6. 确定执行下面指令后 C、Z、DC 标志位的状态：

```
MOVLW  67H
ADDLW  99H
```

2.5 PIC 数据格式和伪指令

本节将介绍 PIC 汇编器支持的常用数据格式和伪指令。

2.5.1 PIC 数据类型

PIC 微控制器只有 8 位数一种数据类型，而且所有寄存器也是 8 位的。所有大于 8 位(00 到 FFH，或者十进制的 0 到 255)的数据在被 CPU 处理前都会被分解。关于如何处理大于 8 位数据的例子，请参阅第 5 章。PIC 能接受的数据可以是正数或者负数。带符号数的处理方法也请参阅第 5 章。而位寻址数据将在第 4 章和第 6 章介绍。

2.5.2 数据格式描述

在 PIC 汇编器里，有 4 种方法用来表示字节数据。而数据可以是十六进制、二进制、十进制，或者是 ASCII 码形式的。下面将逐一地用例子加以说明。

1. 十六进制数

用来表示十六进制数的方法有 4 种。

(1) 可以在数字后加 h (或者 H)，如 MOVLW 99H。

(2) 在数字前加 0x(或者 0X)，如 MOVLW 0x99。

(3) 在数字的前后什么都不加，如 MOVLW 99。

(4) 在数字上打上单引号，并且在前面加 h，如 MOVLW h '99'。

上面的 4 种方法在本书中都会用到。尽管在实际应用中通常只会用到其中的一种方法，但是读者必须习惯使用它们。值得注意的是，当使用 99H 的时候，PIC 汇编器会发出警告(但不是错误报告)，因为它已知这是一个十六进制数，所以无需说明。在编写汇编程序时，这对程序员是一个很好的提醒。

61

下面是用十六进制数写的几行代码：

```
MOVLW 25      ;WREG = 25H
ADDLW 0x11    ;WREG = 25H + 11H = 36H
ADDLW 12H     ;WREG = 36H + 12H = 48H
ADDLW H'2A'   ;WREG = 48H + 2AH = 72H
ADDLW 2CH     ;WREG = 72H + 2CH = 9EH
```

下面的写法是无效的：

```
MOVLW E5H     ;invalid, it must be MOVLW 0E5H
ADDLW C6      ;invalid, it must be ADDLW 0C6
```

注意，在上面的最后两条指令里，如果数值是以十六进制的 A~F 开头的，那么必须在它的前面加 0。不过，下面的书写是有效的：

```
MOVLW 0F     ;valid, WREG = 0FH (or 00001111 in binary)
```

2. 二进制数

在 PIC 汇编器中，二进制数只有一种描述格式，如下所示：

```
MOVLW B'10011001' ;WREG = 10011001 or 99 in hex
```

小写b也是允许的。这同其他的汇编器(如8051和x86)有所不同。下面是如何使用二进制数的例子。

```
MOVLW B'00100101' ;WREG = 25H
ADDLW B'00010001' ;WREG = 25H + 11H = 36H
```

3. 十进制数

对于十进制数,PIC汇编器提供了两种描述格式,其中一种是:

```
MOVLW D'12'        ;WREG = 00001100 or 0C in hex
```

小写d也是允许的。这一点与其他汇编器(如8051和X86)不同。那些汇编器简单地使用十进制数(如12)来定义十进制数,而不在数字前后加任何标志。然而,PIC汇编器会将12默认为十六进制数。下面是如何使用十进制数的例子。

```
MOVLW D'37' ;WREG = 25H (37 in decimal is 25 in hex)
ADDLW D'17' ;WREG = 37 + 17 = 54 where 54 in dec is 36H
```

在一些PIC微控制器的应用资料中,还会看到十进制数的另一种表示形式“.数字”,如下面的例子所示。

62

```
MOVLW .12          ;WREG = 00001100 = 0CH = 12
```

4. ASCII字符

在PIC汇编器里,使用ASCII码时要加上字母A作为标识,如下所示。

```
MOVLW A'2'  ;WREG = 00110010 or 32 in hex (See Appendix F)
```

小写a也是允许的。再一次强调,这一点与其他汇编器(如8051和x86)是不同的。在其他汇编器里,单引号是用于标记单个ASCII码字符的,而双引号是用于标记字符串的。下面是如何使用ASCII字符的例子。

```
MOVLW A'9'  ;WREG = 39H, which is hex number for ASCII '9'
ADDLW A'1'  ;WREG = 39H + 31H = 70H
            ;(31 hex is for ASCII '1')
MOVLW '9'   ;WREG = 39H another way for ASCII
```

要定义ASCII字符串(多于一个字符),可使用DB(定义字节)指令。关于DB的用法将在第6章中介绍。

2.5.3 汇编伪指令

指令是告诉CPU要做什么,而伪指令是指示汇编器工作的。例如,MOVLW和ADDLW指令是CPU要执行的命令,而EQU、ORG和END就是汇编语言的指令。下面将介绍PIC中最为广泛使用的伪指令和用法。

1. EQU

该指令用于定义一个常数或者是固定地址。EQU伪指令并不是给数据分配存储空间,

而是给常数"贴上"一个数据或者地址标签。当标签在程序中出现的时候,它代表的常量就会替代标签值。下面是使用 EQU 作为计数常量,然后将常量赋予 WREG 寄存器:

```
COUNT EQU    0x25
      ...            ....
      MOVLW        COUNT ;WREG = 25H
```

当执行上面的 MOVLW COUNT 命令时,寄存器 WREG 会被赋值为 25H。使用 EQU 命令的好处是什么呢? 假设一个常量(固定值)在整个程序里都用到,而且程序员又想在所有位置修改它的数值。如果使用了 EQU,那么程序员只要修改一处,汇编器就会自动地修改程序里它出现的数值,而不是在程序里逐一地找出它的位置来修改。

2. SET

SET 伪指令是用来定义一个常量或者固定地址的。在这个意义上,SET 和 EQU 是相同的。唯一的不同点是,SET 指令分配的值有可能被再次分配。

2.5.4 使用 EQU 做定值分配

为更多了解使用 EQU 分配定值的用法,请看下面的程序。

```
             ;in hexadecimal
DATA1 EQU    39            ;hex data is the default
DATA2 EQU    0x39          ;another way for hex
DATA3 EQU    39H           ;another way for hex (redundant)
DATA4 EQU    H'39'         ;another way for hex
DATA5 EQU    h'39'         ;another way for hex

             ;in binary
DATA6 EQU    b'00110101' ;binary (35 in hex)
DATA7 EQU    B'00110101' ;binary (35 in hex)

             ;in decimal
DATA8 EQU    D'28'         ;decimal numbers (1C in hex)
DATA9 EQU    d'28'         ;second way for decimal

             ;in ASCII
DATA10  EQU  A'2'          ;ASCII characters
DATA11  EQU  a'2'          ;another way for ASCII char
DATA12  EQU  '2'           ;another way for ASCII char
```

通常,用 DB 伪指令来为定值数据(如 ASCII 码字符串)分配代码 ROM 空间。更多例子请参阅第 6 章。

2.5.5 使用 EQU 做 SFR 地址分配

EQU 伪指令也广泛地用于 SFR 地址分配,请看下面的程序。

```
COUNTER EQU 0x00   ;counter value 00
PORTB EQU 0xFF6    ;SFR Port B address
MOVLW COUNTER      ;WREG = 00H
MOVWF PORTB        ;Port B now has 00 too
INCF  PORTB, F     ;Port B has 01
INCF  PORTB, F     ;increment Port B (Port B = 02)
INCF  PORTB, F     ;increment Port B (Port B = 03)
```

以上都是PIC18系列的指令。若使用的端口B地址不同的其他系列PIC控制器(如PIC16F),则可以修改端口B的EQU地址并重新汇编程序,然后运行程序。

```
COUNTER EQU 0x00   ;counter value 00
PORTB EQU 0x07     ;Port B addr in PIC16F
MOVLW COUNTER      ;WREG = 00H
MOVWF PORTB        ;Port B now has 00 too
INCF  PORTB, F     ;Port B has 01
INCF  PORTB, F     ;Port B has 02
INCF  PORTB, F     ;Port B has 03
...
```

64

2.5.6　使用EQU做RAM地址分配

EQU伪指令的另一个惯常用法是为文件寄存器的通用存储区域分配地址空间。下面使用EQU命令重写以前的例子。

```
MYREG EQU 0x12      ;assign RAM loc to MYREG

MOVLW  0            ;clear WREG (WREG = 0)
MOVLW  MYREG        ;clear MYREG (loc 12H has 0)
MOVLW  22H          ;WREG = 22H
ADDWF  MYREG, F     ;MYREG = WREG + MYREG
ADDWF  MYREG, F     ;MYREG = WREG + MYREG
ADDWF  MYREG, F     ;MYREG = WREG + MYREG
ADDWW  MYREG, F     ;MYREG = WREG + MYREG
```

因项目设计需要改用PIC芯片而修改地址时,该命令尤其有用。访问RAM地址时,使用名字显得比使用数字更方便。

下面的程序是把数值9放入RAM地址0~4,然后求和,并把结果放入地址10H。

```
MYVAL EQU   9      ;MYVAL = 9
R0    EQU   0      ;assign RAM addresses to R0
R1    EQU   1      ;to R1
R2    EQU   2
R3    EQU   3
R4    EQU   4
SUM   EQU   10H

MOVLW MYVAL        ;WREG = 9
MOVWF R0           ;RAM loc 0 has 9
MOVWF R1           ;RAM loc 1 has 9
MOVWF R2           ;RAM loc 2 has 9
MOVWF R3           ;RAM loc 3 has 9
MOVWF R4           ;RAM loc 4 has 9
MOVLW 0            ;WREG = 0
ADDWF R0, W        ;WREG = R0 + WREG
ADDWF R1, W        ;WREG = R1 + WREG
ADDWF R2, W        ;WREG = R2 + WREG
ADDWF R3, W        ;WREG = R3 + WREG
ADDWF R4, W        ;WREG = R4 + WREG
MOVWF SUM
```

1. ORG伪指令

ORG伪指令用于确定起始地址。它可以用作代码或者数据。ORG必须后接十六进制数。

65

2. END 伪指令

另一个重要的伪指令是END伪指令。它向汇编器表明源文件的结束。END伪指令是PIC程序的最后一行,即END伪指令以后的源代码都将被汇编器忽略。

3. LIST 伪指令

与ORG和END这种适用于所有汇编器的指令不同,LIST指令是PIC汇编语言里独有的。它向汇编器表明程序面向的具体PIC芯片信号。例如:

```
LIST P=18F458
```

上述指令告诉PIC汇编器:程序是为PIC18F458微控制器编写的。使用LIST伪指令可指明目标芯片的型号。

4. #include 伪指令

#include伪指令告诉PIC汇编器程序所需的特定PIC芯片的库文件。

5. _config 伪指令

_config伪指令告诉编译器目标PIC芯片的配置位。正确使用_config伪指令是很重要的,否则会导致芯片不能正常工作。PIC设备每次启动时都会读取配置位,然后把状态存入地址300000H。为了使得配置简易化,Microchip公司定义了许多_config伪指令符号。这些符号存放在所采用PIC设备的.INC文件里。更多详情请参阅第8章。

6. radix 伪指令

伪指令radix用来说明系统使用的是十六进制还是十进制的数制。若不使用radix伪指令,则系统默认为十六进制。若使用指令 redix dec,则默认的数制将变成十进制,凡是没有指明格式的数值都将被当作十进制数,而不是十六进制数。

2.5.7 汇编语言的标签规则

选用标签非常有用,程序员可以让程序更加易读和可维护。使用标签时需要遵守一定的规则。第一,每个标签必须唯一。汇编语言的标签名由大小写字母、数字0~9以及特殊的符号[如问号(?)、点(.)、@、下划线(_)、美元符($)等]组成。标签名的首字母必须是英文字母。换而言之,标签名的首字母不能是数字。每个汇编器都有自己的保留字,标签名不能是系统的保留字。首要的保留字就是指令助记符。例如,MOVWL和ADDWL都是保留的,因为它们是指令助记符。除助记符外,还有一些其他的保留字。查阅汇编器可知其保留字列表。

2.5.8 复习题

1. 给出PIC汇编语言中十六进制数的3种表示格式。
2. 分别给出十进制数99在PIC汇编语言里的十六进制、十进制和二进制形式。
3. 使用EQU伪指令定义常数的好处是什么?
4. 给出下面指令中所用到的十六进制数。
   (a) ASC_DATA EQU A '4'　　(b) MY_DATA EQU B '00011111'
5. 给出下面程序的WREG寄存器的内容。

```
MYCOUNT    EQU   15
MOVLW      MYCOUNT
```

6. 给出下面程序的 fileReg 0x20 的值。

```
MYCOUNT    EQU   0x95
MYREG      EQU   0x20
MOVLW      MYCOUNT
MOVWF      MYREG
```

7. 给出下面程序的 fileReg 0x63 的值。

```
MYDATA    EQU  D'12'
MYREG     EQU  0x63
FACTOR    EQU  0x10
MOVLW     MYDATA
ADDLW     FACTOR
MOVWF     MYREG
```

2.6 PIC 汇编语言编程

本节将讨论汇编语言格式以及同汇编语言编程有关的、广泛使用的术语。

CPU 仅以二进制形式就可以高速运行。但是对于人类来说，要为计算机编制二进制程序则是相当乏味的。只有 0 和 1 组成的程序，称为机器语言。在计算机的早期，程序都是由程序员用机器语言编写的。虽然十六进制的效率较二进制表示要高很多，但是机器代码的处理对人类而言还是繁琐的。最终，汇编语言问世了，它不仅用伪代码代替机器指令代码，还增加了其他特性，提高了编程速度，降低了代码出错率。助记符常用于计算机科学和工程学领域，意指相对容易记忆的代码或缩写词。汇编语言程序必须由被称为汇编器的程序编译成机器代码。汇编语言是一种低级语言，因为它直接面向 CPU 的内部结构。要编写汇编语言程序，程序员必须熟知 CPU 的所有寄存器和大小，以及其他技术细节。

现在，人们可以使用许多不同的编程语言，如 BASIC、Pascal、C、C++、Java 等。这些语言被称为高级语言，因为程序员并不需要了解 CPU 的内部细节。汇编器旨在把汇编程序编译成机器代码(有时被称作目标代码或者操作码)，而高级语言要翻译成机器代码，则需要借助于编译器。例如，用 C 语言写的程序必须使用 C 编译器来翻译成机器语言。下面将介绍 PIC 汇编语言的格式。

67

2.6.1 汇编语言结构

汇编语言程序由多行的汇编语言指令构成。一条汇编语言指令由一个伪代码构成，可带一个或者两个操作数。操作数是指参与运算的数据，而助记符是对 CPU 要执行的命令，告诉 CPU 如何处理操作数。

如程序 2-1 所示，汇编语言程序就是一系列的汇编指令(如 ADDLW 和 MOVWF，或者伪指令语句)。当指令告诉 CPU 做什么运算时，伪指令将向汇编器给出指示。例如，在例 2-1 中，当要求 CPU 执行 MOVWF 和 ADDLW 指令时，ORG 和 END 对编译器是伪指令。伪指令 ORG 告诉汇编器把操作码放在内存的地址 0，而伪指令 END 则向汇编器表明源代码的结尾。换而言之，前者伪指令用于表征程序的起点，而后者伪指令用于表征程序的结尾。

一条汇编语言指令由 4 字段组成：

```
[label]    mnemonic   [operands]   [;comment]
```

括号表示该字段是可选的,不是每一条指令都必需。括号是不需要输入的。关于上面的指令格式,需要注意以下几点。

(1) 标签字段允许程序用名字指向一行代码。标签字段的长度不能超过一定的字符数,具体长度因汇编器而异。

(2) 汇编语言助记符(指令)和操作数一起执行程序的实际工作,并实现程序的功能。举个例子,有下面的汇编语句:

```
MOVLW 55H
ADDLW 67H
```

ADDLW 和 MOVLW 都是助记符,用于生成操作码;55H 和 67H 都是操作数。除了助记符和操作数,这两个字段还包括汇编器的伪指令。要记住,与被翻译成机器代码(操作码)供 CPU 执行的指令相反,伪指令并不产生任何机器代码(操作码),并且只供汇编器使用。如程序 2-1 所示,命令 ORG 和 END 是伪指令的例子。这类的伪指令的更多细节将在以后的章节中讨论。

68

(3) 注释字段以分号";"开头。注释可以放在一行代码的后面,也可以独立成行。尽管汇编器会忽略注释,但注释对程序员来说是不可或缺的。虽然注释是可选的,但是仍然推荐使用它,因为它使人们更容易读懂程序。

(4) 注意程序 2-1 中标签字段上的标签 HERE。在 GOTO 语句里,它告诉 PIC 在此循环等待。如果系统是一个监控程序,那么不需要该行语句,应该将其从程序中删除。2.7 节将介绍如何生成一个准备就绪程序。

程序 2-1 汇编语言程序范例

```
      ;PIC Assembly Language Program To Add Some Data.
      ;store SUM in fileReg location 10H.
SUM   EQU   10H             ;RAM loc 10H for SUM

      ORG   0H              ;start at address 0
      MOVLW 25H             ;WREG = 25
      ADDLW 0x34            ;add 34H to WREG
      ADDLW 11H             ;add 11H to WREG
      ADDLW D'18'           ;W = W + 12H = 7CH
      ADDLW 1CH             ;W = W + 1CH = 98H
      ADDLW B'00000110'     ;W = W + 6 = 9EH
      MOVWF SUM             ;save the SUM in loc 10H
HERE  GOTO  HERE            ;stay here forever
      END                   ;end of asm source file
```


2.6.2 复习题

1. 使用伪指令的目的是什么?
2. ________会被汇编器翻译成机器代码,而________则不会。
3. 判断对错:汇编语言是高级语言。
4. 下面哪些指令将产生操作码?请列出所有的操作码。

(a) MOVLW 25H　(b) ADDLW 12　(c) ORG 2000H　(d) GOTO HERE

5. 伪指令也被称作________。

6. 判断对错：汇编器伪指令不是供 CPU 使用的，它们只是对汇编器起引导作用。

69

7. 在第 4 题中，哪个指令是汇编器指示语句？

2.7 汇编和连接 PIC 程序

到此，汇编语言程序的基本格式已经介绍过了。接下来的问题是：程序是怎样生成、汇编和准备运行的呢？生成一个可执行程序的主要步骤可归纳如下（见图 2-8）。

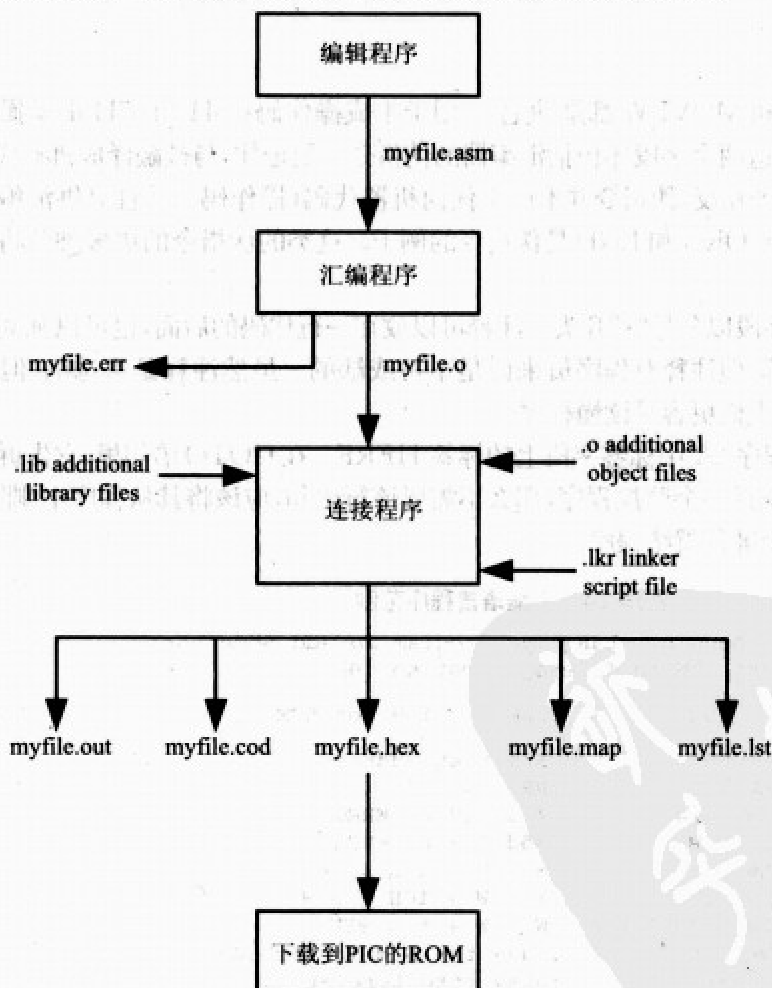


图 2-8　创建程序的步骤

(1) 首先使用文档编辑器录入如程序 2-1 所示的程序源代码。对于 PIC 微控制器，可以使用 MPLAB IDE，它是集文档编辑器、汇编器、连接器、模拟器和更多功能于一体的软件包，是一款出色的支持所有 PIC 芯片开发的免费软件。很多的编辑器和文字处理器都可以用来生成和编辑程序。常用的编辑器有 MS-DOS 自带的 EDIT 软件和所有 Windows 系统都会自带的记事本软件。注意，所有的编辑器都要能生成 ASCII 码文档。对于汇编程序，文件名要遵守 DOS 的惯例，但是源文件还要有 asm 的扩展名。源文件的 asm 扩展名将在第(2)步的汇

编器中用到。

(2) 将在第(1)步中所生成的程序代码 asm 源文件提交给 PIC 汇编器。汇编器把指令转换成机器码。同时,汇编器还会生成一个目标文件和一个错误报告文件。目标文件的扩展名是 o。错误报告文件指出了程序的语法错误和所在行号,其扩展名为 err。错误报告文件可以用任何文档编辑器浏览。

70

(3) 编译器的第 3 步是连接。连接程序输入一个或多个目标文件,然后输出一个十六进制文件、一个列表文件、一个映像文件、一个临时目标文件和一个调试文件。十六进制文件的扩展名为 hex,列表文件的扩展名为 lst,映像文件的扩展名为 map,临时目标文件的扩展名为 out,调试文件的扩展名为 cod。在成功连接后,准备好的十六进制文件就可以烧录到 PIC 程序 ROM 和下载到 PIC 调试装置中。更多详细请参阅第 8 章。

MPLAB IDE 是一个基于 Windows 系统的程序,在文件录入后将第(2)步和第(3)步合成为一步。

2.7.1 关于 asm、err 和目标文件的更多信息

asm 文件,也被称作源文件,必须以 asm 作为扩展名。如前所述,该文件由文档编辑器(如 MS-DOS 自带的 EDIT 软件和 Windows 自带的记事本程序)创建。很多汇编器都带有文件编辑器。汇编器把 asm 文件的汇编语言转换成机器语言,并生成 o(目标)文件。PIC 汇编器还会生成目标文件和错误报告文件。像前面所说的那样,目标文件以 o 为扩展名。在标准程序里,使用连接器可以把很多目标文件连接起来生成一个可运行的十六进制文件,这将在第 6 章中介绍。但是,在连接器生成一个可运行的程序之前,必须确保程序没有错误。PIC 汇编器提供以 err 为扩展名的错误报告文件,以备检查程序中的语法错误。在语法错误纠正之前,连接器是不会连接程序的。工程师可以把错误报告文件打印出来或者使用 Notepad(记事本)来检查错误。然后再返回到 asm 文件,在汇编之前更正所有存在的错误。下面就是一个错误报告文件的例子。

程序 2-1 PIC 汇编源代码例子(asm 文件)

```
;PIC Assembly Language Program To Add Some Data.
;store sum in fileReg location 10H.

SUM     EQU     10H             ;RAM loc 10H for sum

        ORG     0H              ;start at address 0
        MOVLW 25H               ;WREG = 25
        ADDLW 0x34              ;add 34H to WREG
        ADDLW 11H               ;add 11H to WREG
        ADDLW D'18'             ;W = W + 12H = 7CH
        ADDLW 1CH               ;W = W + 1CH = 98H
        ADDLW B'00000110'       ;W = W + 6 = 9EH
        MOVWF SUM               ;save the sum in loc 10H
HERE    GOTO HERE               ;stay here forever
        END                     ;end of asm source file
```

71

程序 2-1 PIC 错误报告例子(err 文件)

```
Warning[207] C:\MDEPIC\EXAMPLE 2-1.ASM 6  : Found label after column 1. (R4)
Warning[207] C:\MDEPIC\EXAMPLE 2-1.ASM 13 : Found label after column 1. (movle)
Error[122]   C:\MDEPIC\EXAMPLE 2-1.ASM 13 : Illegal opcode (d)
```

```
Warning[207] C:\MDEPIC\EXAMPLE 2-1.ASM 17 : Found label after column 1. (DEC)
Error[122]   C:\MDEPIC\EXAMPLE 2-1.ASM 17 : Illegal opcode (COUNT)
Warning[203] C:\MDEPIC\EXAMPLE 2-1.ASM 20 : Found opcode in column 1. (movwf)
Warning[207] C:\MDEPIC\EXAMPLE 2-1.ASM 21 : Found label after column 1. (addl)
Error[108]   C:\MDEPIC\EXAMPLE 2-1.ASM 21 : Illegal character (0)
Error[116]   C:\MDEPIC\EXAMPLE 2-1.ASM 29 : Address label duplicated or differ-
ent in second pass (AGAIN)
```

2.7.2 列表文件和映像文件

Lst(列表)和MAP(映像)文件对程序员是非常有用的。列表文件列出了二进制码和程序源代码。MAP文件列出存储器的在用和未用地址的分布情况。这两个文件可以使用诸如记事本等编辑器在显示器上查看,也可以通过打印机打出副本。程序员通常使用列表文件和MAP文件来确认争取的系统设计。

程序2-1 列表文件

```
LOC    OBJECT CODE LINE SOURCE TEXT          VALUE

                   00001
                   00002 ;PIC Asm Language Program To Add Some Data
                   00003 ;store SUM in fileReg location 10H
  00000010         00004 SUM EQU 10H          ;RAM loc 10H for Sum
                   00005
000000             00006 ORG 0H               ;start at address 0
000000 0E25        00007 MOVLW 25H            ;WREG = 25
000002 0F34        00008 ADDLW 0x34           ;add 34H to WREG
000004 0F11        00009 ADDLW 11H            ;add 11H to WREG
000006 0F12        00010 ADDLW D'18'          ;W = W + 12H = 7CH
000008 0F1C        00011 ADDLW 1CH            ;W = W + 1CH = 98H
00000A 0F06        00012 ADDLW B'00000110'    ;W = W + 6 = 9EH
00000C 6E10        00013 MOVWF SUM            ;save the SUM in loc 10H
00000E EF07 F000   00014 HERE GOTO HERE       ;stay here forever
                   00015 END                  ;end of asm source file
```

2.7.3 复习题

1. 判断对错:MPLAB、MS-DOS Edit 和 Windows Notepad 文档编辑器都可以产生 ASCII 文件。
2. 判断对错:源文件的扩展名是 asm。
3. 下面的哪个文件可以由文档编辑器生成?
   (a) myprog.asm (b) myprog.o (c) myprog.hex (d) myprog.lst (e) myprog.err
4. 下面的哪个文件由编译器生成?
   (a) myprog.asm (b) myprog.o (c) myprog.hex (d) myprog.lst (e) myprog.err
5. 下面的哪个文件列出语法错误?
   (a) myprog.asm (b) myprog.o (c) myprog.hex (d) myprog.lst (e) myprog.err

2.8 PIC的程序计数器和程序ROM空间

本节将讨论程序计数器(PC)在程序执行过程中扮演的角色,以及代码是如何从ROM里提取和执行的。

2.8.1 PIC 的程序计数器

PIC 微控制器里另外一个重要的寄存器是程序计数器(PC)。CPU 使用程序计数器来指示下一条要执行的指令的地址。当 CPU 从程序存储器里取得操作码时,程序计数器就会自动加 1,从而指向下一条指令。程序计数器的计数范围越宽,则 CPU 访问的程序空间越大。例如,一个 14 位的程序计数器可以访问的最大代码地址空间是 16 KB(2^{14}=16K),地址范围为 0000~3FFF。PIC 的 16F 系列带有 14 位的程序计数器,而 PIC12F 的程序计数器是 12 位的。对于 16 位的程序计数器,代码空间是 64 KB(2^{16}=64K),地址范围为 0000~FFFF。8051 微控制器带有 16 位的程序计数器,而 PIC18 系列微控制器带有高达 21 位的程序计数器。这意味着 PIC18 系列可以访问的程序地址为 000000~1FFFFF,地址空间大小为 2 MB。然而,并非所有的 PIC18 芯片都有 2 MB(2^{21}=2M)的片上 ROM,请参阅表 2-6。PIC16C 系列的 14 位程序计数器的最大代码容量仅仅是 16 KB。为了克服这种主要的限制,PIC 设计人员不得不在后来的 PIC16 芯片上引入翻页功能。设计人员从中吸收了经验教训,所以 PIC18 系列的程序计数器扩展到了 21 位,解决了代码容量限制的问题,如图 2-9 所示。2 MB 的代码空间对于以后很多年的发展都是够用的。表 2-6 中的数据摘自 Microchip 的网站。

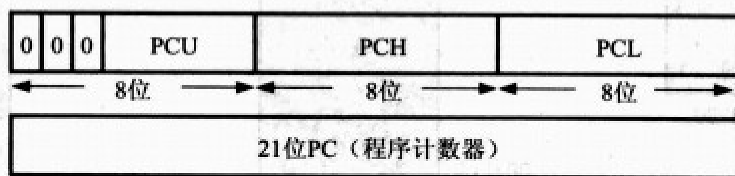


图 2-9 PIC 的程序计数器

表 2-6 PIC18 的片上 ROM 大小和地址空间

	片上代码 ROM (B)	代码地址范围(十六进制)
PIC18F2220	4 K	00000~00FFF
PIC18F2410	16 K	00000~03FFF
PIC18F458	32 K	00000~07FFF
PIC18F6680	64 K	00000~0FFFF
PIC18F8722	128 K	00000~1FFFF

2.8.2 PIC18 系列 ROM 的内存分配

正如前面看到的,有些系列的芯片(如 PIC18F2220)只有几 KB 的片上 ROM 空间,而有些芯片(如 PIC18F6680)则有 64 KB 的 ROM。PIC18F458 有 32 KB 的片上 ROM。要记住的是,PIC 系列芯片的操作码空间不能超过 2 MB,因为 PIC 的程序计数器只有 21 位(地址范围是 000000~1FFFFF)。值得注意的是,当程序 ROM 的第一个地址是 000000 时,最后一个地址是不同的,这取决于不同芯片的 ROM 大小,如图 2-10 所示。在 PIC18 系列中,PIC18F2220 有 4 KB的片上 ROM,其地址范围是 0000~000FFFH。因此,片上 ROM 的第一个地址是 000000,最后一个地址是 0FFFH。例 2-11 演示了如何计算这些地址。

例 2-11 找出下面 PIC 芯片的 ROM 地址：

(a) PIC18F2220，大小 4 KB

(b) PIC18F2410，大小 16 KB

(c) PIC18F458，大小 32 KB

解：

(a) 对于 4 KB 的片上 ROM 空间，共有 4096 B(4×1024=4096)。它的地址分布为 0000～0FFFH。注意，地址总是从 0 开始的。

(b) 对于 16 KB 的片上 ROM 空间，共有 16 384 B(16×1024=16 384)。它的地址分布为 0000～3FFFH。

(c) 对于 32 KB 的片上 ROM 空间，共有 32 768 B(32×1024=32 768)。它的地址分布为 0000～7FFFH。

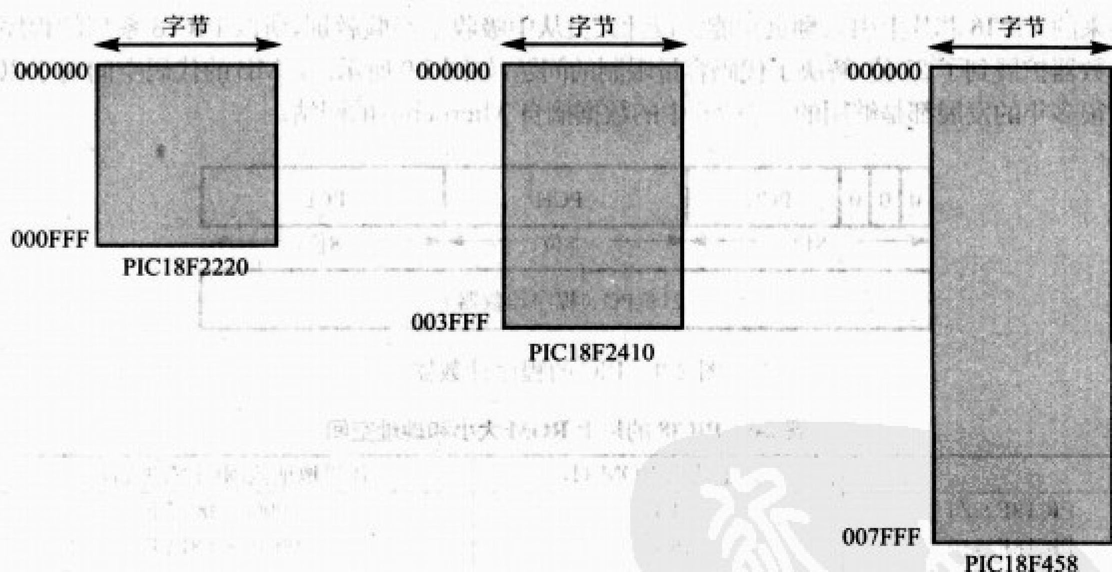


图 2-10 PIC18 片上程序(代码)ROM 地址范围

2.8.3 通电时 PIC 的启动

对于任何的微控制器(或者微处理器)都有这样一个问题：当通电时，CPU 从哪个地址开始启动？答案是：每类微控制器都是不同的。就 PIC 单片机来讲，不论何种系列和型号，在通电时微控制器都是从地址 0000 开始启动的。正如第 8 章所讲的，通电意味着给 RESET 引脚施加电压。换而言之，当 PIC 单片机通电时，PC(程序计数器)的值就是 00000。这意味着，第一个操作码必须要存放在 ROM 的 0000H 地址(见图 2-11)。基于这个理由，PIC 系统的第一个操作码必须烧录在 ROM 的 00000H 地址，因为 PIC 单片机启动时在这里寻找第一条指令。正如前面所述，可以在源代码程序中使用 ORG 命令来完成该步骤。下面将讨论在取指令和执行程序中程序计数器是如何一步一步地工作的。

2.8.4 在程序 ROM 里放置代码

为了更好地理解程序计数器在读取和执行程序时扮演的角色，接下来讨论程序计数器在每一条指令的读取和执行时的操作。首先，让我们来回顾例子例程的列表文件，看看代码在 PIC 芯片的 ROM 里是如何存放的。正如我们看到的，每条指令的操作数和操作码都清晰地罗列在列表文件的左边。

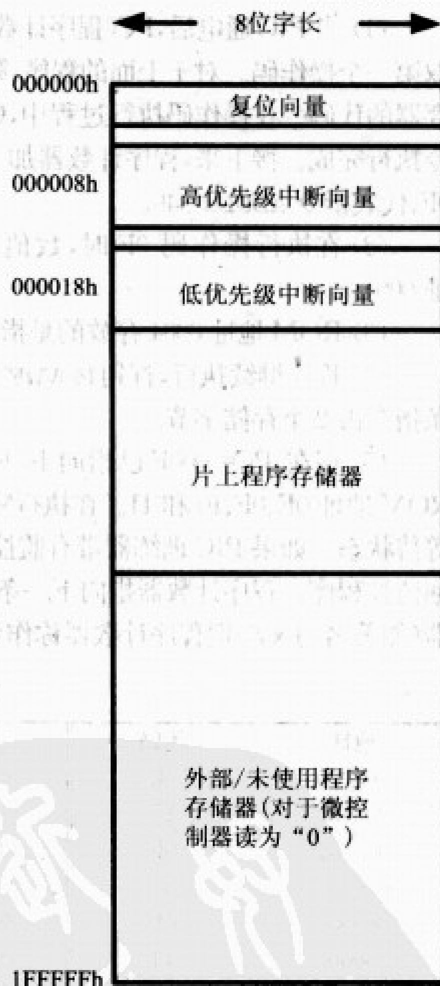


图 2-11 PIC 程序 ROM 空间

当程序被烧录到 PIC 系列(如 PIC18F452 和 PIC18F458 芯片)的 ROM 中时，操作码和操作数都会被放置在从 0000 开始的 ROM 存储器中，如程序 2-1 的列表文件所示。

从列表可知，地址 0000 的内容是 0E，就是把数值传送至 WREG 的操作码；地址 0001 的内容是要传送至 WREG 的操作数(此处为 25H)。因此，指令 MOVLW 25H 的机器码是 0E25，其中 0E 是操作码，25 是操作数。类似地，机器码 0F34 被放置在地址 0002 和 0003，分别表示指令 ADDLW 34H 的操作码和操作数。同样，机器码 0F11 被放置在地址 0004 和 0005，分别表示指令 ADDLW 11H 的操作码和操作数。内存地址 0006 的内容为操作码 0F，也就是指令 MOVLW 的操作码，地址 0007 的值是 12，代表指令 ADDLW D'18' 的十进制操作数 18。指令 ADDLW 1CH 的操作码被放置在地址 0008，操作数 1CH 位于地址 0009。地址 000A 和 000B 存放 ADDLW B '00000110' 的操作码和操作数。指令 MOVWF SUM 的操作码存放地址 000C，而操作数的地址 10H 存放在地址 0000D。GOTO HERE 的操作码和目标地址依次存放在地址 00000E、00000F、000010 和 000011。在该程序中，除 GOTO 是 4 B 的指令外，其余所有的指令都是 2 B 指令。其中的奥秘将留在本小节的最后解释。

ROM 地址	机器语言	汇编语言
00000	0E25	MOVLW 25H
00002	0F34	ADDLW 34H
00004	0F11	ADDLW 11H
00006	0F12	ADDLW D'18'
00008	0F1C	ADDLW 1CH
0000A	0F06	ADDLW B '00000110'
0000C	6E10	MOVWF SUM
0000E	EF07 F000	HERE GOTO HERE

75
76

2.8.5　程序的逐字节执行

假设上面介绍的程序已被烧录到PIC18芯片的ROM中，下面将一步一步地阐述PIC通电后的动作。

(1) 当PIC通电后，PC(程序计数器)的值为00000，将从程序ROM的00000地址开始读取第一个操作码。对于上面的程序，第一个操作码是0E，表示把一个操作数传送至WREG寄存器的代码。在操作码执行过程中，CPU把数值25H赋给WREG寄存器。至此，第一条指令执行完成。接下来，程序计数器加1，指向地址00002(PC=00002)，该地址的内容是操作码0F，代表指令ADDLW 34H。

(2) 在执行操作码0F时，数值34H被加到了WREG中。然后，程序计数器指向地址0004。

(3) ROM地址0004存放的是指令ADDLW 11H的操作码。该指令执行完毕后，PC=0006。

(4) 程序继续执行，直到MOVWF SUM被读取和执行。注意，上面的指令都是2 B的，即每条指令占2个存储字节。

(5) 现在，PC=000E已指向下一条指令——GOTO HERE。这是一条4 B的指令，它占用ROM地址0E、0F、10和11。在执行完该指令后，PC=0000E。于是，程序进入一个无限循环的等待状态。如果PIC训练器带有监控程序，那么程序就不必使用GOTO指令，程序会自己返回监控程序。程序计数器指向下一条要执行的指令的事实，很好地回答了为什么有的微处理器(如著名的x86)把程序计数器称作指令指针。

程序2-1的ROM内容

地址	代码	地址	代码	地址	代码
000000	0E	000007	12	00000D	10
000001	25	000008	0F	00000E	07
000002	0F	000009	1C	00000F	EF
000003	34	00000A	0F	000010	00
000004	0F	00000B	06	000011	F0
000005	11	00000C	6E	000012	
000006	0F				

2.8.6　PIC18 ROM数据宽度

正如本节前面所讨论的，存储代码的微处理器内存是字节可访问的，而且受程序计数器的控制。换而言之，地址空间的每个位置只能存储1 B。例如，16条地址线对应着2^{16}个地址，即地址范围从0000到FFFFH共64 KB的存储空间。8位数据的CPU每次只能读取一个字节，这就是用于第一台IBM个人电脑和苹果电脑的实例。为了使CPU能读取更多信息(代码或数据)，可以将数据总线的宽度增加到16位。1984年，IBM的PC AT就应用了这种方法。为了更进一步地提升性能，Intel公司把386处理器的数据总线宽度增加到32位，而将奔腾处理器的数据总线宽度增加到64位。在某种意义上，数据总线犹如高速公路上的车道，每个车道就是8位宽度。车道越多，CPU就能有更多的信息处理。对于PIC18，代码ROM和CPU

之间的内部数据总线是 16 位，如图 2-12 所示。因此，在使用 16 位宽度的数据时，64KB 的 ROM 空间可表示为 32KB×16。同样地，PIC18 所有 2MB 的程序地址空间，可表示为 1MB×16。增加程序 ROM 和 CPU 之间的数据宽度是 PIC 设计人员增强 PIC18 系列单片机处理能力的另一种方法。把代码 ROM 数据宽度设计成 16 位的另外一个目的是为了同 PIC18 的指令长度相匹配，因为大多数的指令都是 2 字节长的。这样，CPU 每次访问程序 ROM 时都能从中读取一条指令。于是，指令的读取就成了单周期操作，这将在下一章讨论指令时序时更详细地介绍。

PIC18 的设计人员将所有指令都设计成为 2B 或者 4B；没有像 x86 和 8051 那样设计的 1B 或 3B 指令。这是 RISC 结构理论的组成部分，将在下一节讨论。要注意的是，并非所有 PIC 微控制器的程序 ROM 都具有 16 位宽度。PIC16 的 ROM 数据长度为 14 位，而 PIC12 的 ROM 数据宽度则是 12 位。另外，还要注意的是，PIC 微控制器的文件寄存器里数据存储器 SRAM 的长度为 8 位，和程序 ROM 一样，也是字节可寻址的。

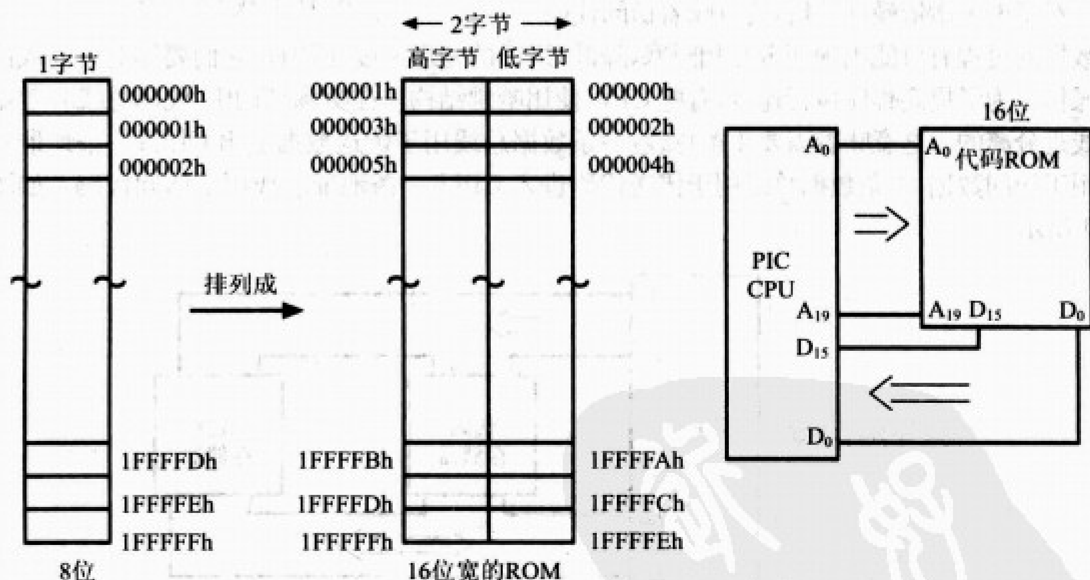


图 2-12 PIC 程序 ROM 数据宽度

little-endian 与 big-endian

让我们来研究如图 2-13 所示的 PIC18 ROM 的代码存放。低字节代码放在存储器的低地址，高字节代码放在高地址。为了区别 big-endian 规范，这个规则被称为 little-endian 规范。术语 big-endian 和 little-endian 来源于《格利佛游记》里关于怎么打开鸡蛋的故事：是从大头打开还是小头打开。big-endian 的方法是将高字节代码放在低地址，而 little-endian 的方法是高字节代码放在高地址，低字节代码放在低地址。所有的 Intel 微处理器和很多小型计算机，比如 DEC 的 VAX，都是采用 little-endian 结构。Motorola 公司的 Freescale 微处理器（用于 Macintosh 机）和一些大型机采用 big-endia 结构。两者的区别看起来就跟从大头还是小头打开鸡蛋一样微不足道，但是在将软件从一个阵营的机器移植到另一个阵营机器上运行时却遇到了麻烦。有些微处理器[例如来自 IBM 或者 Freescale(Motorola)的 PowerPC]可以让软件

设计人员选择 little-endian 规范还是 big-endian 规范。

字地址	高字节	低字节
000000h	0Eh	25h
000002h	0Fh	34h
000004h	0Fh	11h
000006h	0Fh	12h
000008h	0Fh	1Ch
00000Ah	0Fh	06h
00000Ch	6Eh	10h
00000Eh	EFh	07h
000010h	0Fh	00h

图 2-13 与程序 2-1 列表文件对应的 PIC 程序 ROM 内容

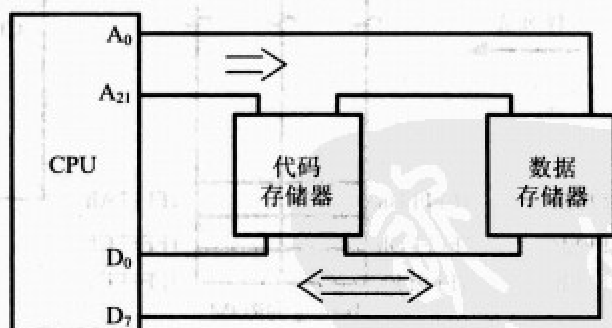
2.8.7 PIC 的哈佛结构

任何微处理器都必需具备存储程序(代码)和数据的存储空间。正如已经学习到的,PIC 也不例外,带有代码 ROM 空间和数据 RAM(文件寄存器)空间。代码为 CPU 提供指令,而数据则提供要处理的信息。CPU 使用总线(线路走线)来访问代码 ROM 和数据 RAM 空间。早期的计算机使用同一条总线来访问代码和数据。这样的结构通常被称为冯·诺依曼(普林斯顿)结构。对于冯·诺依曼计算机,这意味着访问代码

79

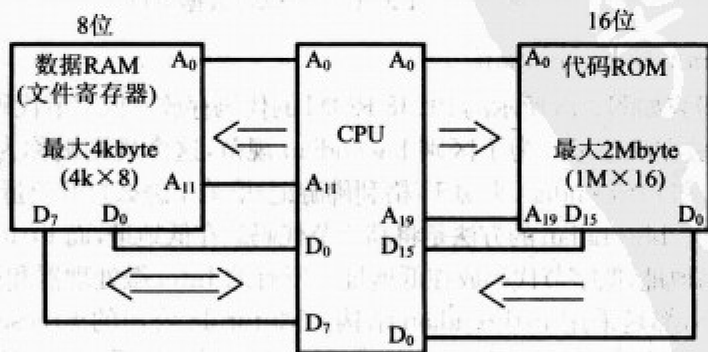
和数据的过程有可能出现两者互相阻塞,降低 CPU 的处理速度,原因是它们要等对方完成读取操作。为了提高程序执行速度,有些 CPU 使用哈佛结构。在哈佛结构中,代码总线和数据总线是分离的。这意味着需要 4 条总线:一条数据总线用于传送数据进出 CPU;一条地址总线用于访问数据;一条数据总线用于传送代码进入 CPU;一条地址总线用于访问代码。如图 2-14 所示。

冯·诺依曼结构



哈佛结构

8位 16位



80

图 2-14 冯·诺依曼结构和哈佛结构的比较

在IC芯片(如微控制器)中,这种结构是容易实现的,因为代码ROM和数据RAM都在芯片上而且它们的距离是微米和毫米级的。但是,要在诸如x86 IBM个人电脑那样的机器上实现哈佛结构,代价是相当昂贵的,因为存储代码和数据的ROM和RAM位于CPU的外部。在主板上分离访问数据和代码的走线将导致电路板很大而且造价高昂。例如,对于一个有64位数据总线和32位地址总线的奔腾微处理器,如果采用冯·诺依曼结构,那么在母板上大约需要100条走线(96条用于地址和数据,还有几条用于读写控制信号等)。但是,如果采用哈佛结构,走线的数量将会倍增到200条。哈佛结构还需要从微处理器本身引出大量的引脚。鉴于上述原因,读者看不到应用在个人电脑和工作站的哈佛结构。这也是有些微控制器(如PIC)在内部采用哈佛结构的原因,但是当需要扩展外部代码或数据空间时,它们依然会采用冯·诺依曼结构。冯·诺依曼结构是在普林斯顿大学发展起来的,而哈佛结构是哈佛大学的成果。

2.8.8 PIC18的指令大小

大家可以回想一下,PIC程序存储器是字节可寻址的,而指令是2B或者4B的。几乎所有的PIC18指令都是2B的。例外的是MOVFF、GOTO和部分其他指令。下面将讨论本章使用过的一些指令的大小和格式,以期对PIC18指令有更深的认识。

2.8.9 MOVLW指令格式

MOVLW是一个2B(16位)的指令。在16位中,前8位是预留给操作码的,其余8位用于从00到FFH的立即数。其指令格式如下。

0000	1110	*kkkk*	*kkkk*

$0 \leqslant k \leqslant FF$

2.8.10 ADDLW指令格式

ADDLW是一个2B(16位)的指令。在16位中,前8位是预留给操作码的,其余8位用于从00到FFH的立即数。其指令格式如下。

0000	1111	*kkkk*	*kkkk*

$0 \leqslant k \leqslant FF$

2.8.11 MOVWF指令格式

MOVWF是一个2B(16位)的指令。在16位中,前8位是预留给操作码的,其余8位用于数据RAM里的文件寄存器地址。操作码的LSB位使用字母*a*来识别是从访问存储器还是其他4096存储器地址访问。如果*a*=0,文件寄存器位于访问存储器里。如果*a*=1,要使用第6章介绍的存储器转换。其指令格式如下。

0110	111a	*ffff*	*ffff*

$0 \leqslant f \leqslant FF$

a=0:用到访问存储器

a=1:用到BSR寄存器指定的访问存储器

请参阅第6章

2.8.12 MOVFF 指令格式

MOVFF 是一个 4B(32 位)指令。在 32 位中,前 16 位分配给了源文件寄存器的操作码和地址,其余 16 位用作目的寄存器的操作码和地址。其指令格式如下。

1100	*ssss*	*ssss*	*ssss*	源地址(f_s)
1111	*dddd*	*dddd*	*dddd*	目的地址(f_d)

$0 \leqslant f_s \leqslant$ FFF

$0 \leqslant f_d \leqslant$ FFF

注意,源地址和目的地址都是指令的一部分。在 PIC18 中,用作文件寄存器地址的有 12 位。这 12 位涵盖了文件寄存器 000 到 FFFH 的所有地址范围,也就是 4096B 数据 RAM 空间。换而言之,MOVFF 指令可以在文件寄存器的范围里任意传送数据,而无需借助 WREG 寄存器,这正如在 2.3 节所看到的。

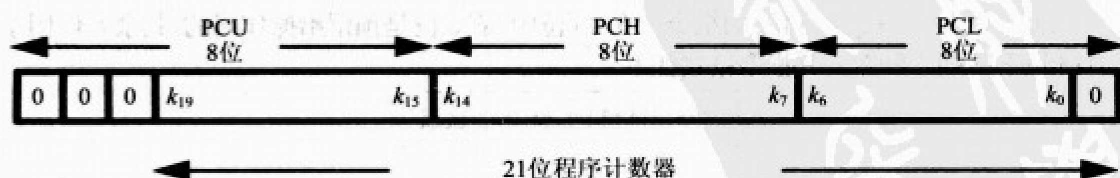
2.8.13 GOTO 指令格式

GOTO 是一个 4B(32 位)指令。在 32 位中,只有 12 位用作操作码,剩下的 20 位用于 GOTO 的目标地址。其指令格式如下。

1110	1111	k_7kkk	$kkkk_0$
1111	$k_{19}kkk$	$kkkk$	$kkkk_8$

$0 \leqslant k \leqslant$ FFFFF

然而,20 位的地址仅能提供 1 MB 的地址空间,而 PIC18 则拥有 2 MB 的 ROM 空间。通过将 GOTO 指令的最低有效位(LSB)置零,这个问题可以得到解决,如下图所示。



预置目标地址的 LSB 为 0 可以使得目标地址为偶地址。正如上一节所介绍的,使用这种置零的方法可以得到设计人员期望的地址,因为所有指令都是 2B 或者 4B。注意,一定要避免在指令中间位置插入零。

2.8.14 从其他微处理器过渡到 PIC18

如果你有其他微处理器或者微控制器的编程经验,那么记住以下关于 PIC18 的知识有助于你更容易地过渡到 PIC18 芯片上来。

(1) PIC18 寄存器里的访问存储器地址范围是 00～7FH,它可以看作一个大量的寄存器集合,只是它们没有其他处理器那样明确的名字。只要我们不使用 SFR、WREG 等的保留字,可以很容易地对寄存器命名。下面是使用 8051 或者其他 RISC 处理器的一个例子:

```
R0      EQU     0
R1      EQU     1
R2      EQU     2
R3      EQU     3
...     ...     ...
```

下面是使用 x86 的例子：

```
BL      EQU     0
BH      EQU     1
CL      EQU     2
CH      EQU     4
DL      EQU     5
DH      EQU     6
```

在上面的两个例子中，可以使用 00～7FH 之间任何地址的文件寄存器。

(2) WREG 同其他微处理器中的累加器很相像。所有的算术运算和逻辑运算操作都必须有 WREG 的参与。

(3) 要把数据传送至文件寄存器或者 SFR，则必须首先把数据传送至 WREG。如前面所述，可以先使用 MOVLW 指令把数值赋给 WREG，然后再用 MOVWF 指令把 WREG 中的数值传送至文件寄存器里的目标地址。换而言之，不能将数值直接传送给 SFR 或者文件寄存器。

2.8.15 复习题

1. PIC18 的程序计数器是________位的。
2. 判断对错：不管程序 ROM 的大小多少，PIC18 系列的每种芯片在通电的时候都是从地址 0000H 开始启动的。
3. PIC18 程序的第一个操作码存放在 ROM 的什么位置？
4. 指令 MOVLW 44H 是一条________位的指令。
5. PIC18F458 的 ROM 地址空间有多大？
6. 指令 GOTO label 是________位的指令。
7. 判断对错：PIC18 的所有指令都是 2 B 或者 4 B 的。

2.9 PIC 的 RISC 结构

微处理器的设计人员提高 CPU 的处理能力有以下 3 种方法。

(1) 提高芯片的时钟频率。该方法的缺点是：频率越高，能耗和热耗就越多。能耗和热耗对于手提设备是一个需要正视的问题。

(2) 使用哈佛结构，通过增加总线的数量来让 CPU 处理更多的信息（代码和数据）。对于 x86 和其他通用微处理器来说，采用哈佛结构的代价是昂贵的，而且不现实，但是对于今天的微控制器来说，这已经不是问题。如前文所述及的，PIC18 采用的就是哈佛结构。

(3) 改变 CPU 内部结构，采用 RISC 结构。

Microchip采用了上面的3种方法来提高PIC18微控制器的处理能力。本节将讨论RISC结构的优点,以及它在PIC18微控制器中的应用。

2.9.1 RISC结构

20世纪80年代初,在计算机设计领域爆发了一场论战,但与其他论战不同的是,这场论战还没有结束。自20世纪60年代以来,不管是大型计算机还是微型计算机,设计人员都把所考虑的许多指令放入到CPU里面。其中一些指令是用来执行复杂操作的。例如,把存储地址的数据相加,然后把结果再放入存储器。自然地,微处理器的设计人员竞相模仿小型计算机和大型计算机设计人员的方法。这些微处理器使用庞大的指令系统,而且在这些指令中大多数都是执行非常复杂的操作,因此,这些微处理器就被称为CISC(Complex Instruction Set Computer,复杂指令集计算机)。根据20世纪70年代所做的研究可知,在这些置入CPU的复杂指令中,许多指令从来没被程序员和编译器使用过。在微控制器中实现大量指令的巨大开销,和在片上需要集成相当数量的用于指令译码器的晶体管的事实,激发了一些设计人员对指令数量的简化和降低的设想。随着这类设想的实现,所设计的处理器就称为RISC(Reduced Instruction Set Computer,精简指令集计算机)。

2.9.2 RISC的特性

下面介绍在PIC18微控制器上实现的RISC的一些特性。

1. 特性1

84

RISC有固定的指令长度。对于CISC微控制器(如8051),指令可以是1 B、2 B,甚至是3 B。例如,下面的8051指令:

```
CLR C                        ;借/进位清零,1 B指令
ADD Accumulator,#mybyte      ;2 B指令
LJMP target_address          ;3 B指令
```

指令长度的不确定性为指令的译码带来相当的困难,因为读入的指令长度是未知的。在RISC结构中,所有指令的长度都是确定的。因此,CPU可以快速地译码指令。打个比方,这就好像跟工人师傅砌大小相同的砖和砌大小不同的砖的区别一样。当然,砌大小相同的砖效率会更高。在过去的一节中,相信读者已看到PIC18是怎样使用2B和少量的4B指令的。

2. 特性2

RISC结构的主要特性之一就是它带有大量的寄存器。所有的RISC结构至少有32个寄存器。在这32个寄存器里,仅有很少的寄存器被指明确定的功能。使用大量寄存器的一个优点是可以避免使用大量的栈来存储参数。虽然RISC处理器也可以使用栈,但并不像CISC处理器那么依赖栈,因为RISC处理器有大量的寄存器可以选用。PIC微控制器使用256B的存储单元作为文件寄存器来满足RISC的这种特性。关于PIC18的栈将在下一章讨论。

3. 特性3

RISC处理器的指令集较小。RISC处理器只有诸如ADD、SUB、MUL、LOAD、STORE、AND、OR、EXOR、CALL和JUMP等基本的指令。有限的指令数量是RISC处理器备受指责的一点,因为这使得汇编语言程序员的工作更乏味、更枯燥,而且无法与CISC汇编语言相

比。因此,RISC 一般更常用在高级语言环境(如 C 语言),而不是汇编语言环境中。有趣的一点是,一些 CISC 的捍卫者称其为完整指令集计算机(complete instruction set computer)而不是复杂指令集计算机(complex instruction set computer),因为它拥有完整的各类指令集。至于有多少指令会被用到和有多常用,则是另外一个问题。RISC 有限的指令导致了程序很长。虽然这些程序占用了更多内存,但这已不成为问题,因为内存很便宜。不过,在 20 世纪 60 年代半导体存储器时代到来之前,CISC 设计人员要尽量在一条指令里实现多个操作来获取更多的利润。PIC16 有 35 条指令,而 PIC18 有 75 条。更多的 PIC18 指令将会在以后的章节中介绍。

4. 特性 4

在此,你不禁会问,既然用 RISC 编程有那么多的麻烦,那它的优点又是什么呢?与 CISC 相比,RISC 最重要的特性是,在一个时钟周期里,处理器能执行 95%以上的指令。尽管在剩下的 5%里有一些指令需要两个时钟周期来执行,但它们可以通过 Juggling 技术(代码调度)在一个时钟周期里完成指令执行。代码调度是编译器经常性的工作。第 3 章将会讨论 PIC18 的指令周期和流水线技术。

5. 特性 5

RISC 处理器有分离的总线,分别用于访问数据和代码。像其他 CISC 计算机一样,所有 x86 处理器有一套地址总线(如 80286 的 A2～A24)和一套数据总线(如 80286 的 D0～D15),用来传送操作码和操作数进出 CPU。为了访问内存中的任何地址,不管是否包含代码或者操作数,系统总是使用相同的地址总线和数据总线。而 RISC 控制器有 4 条总线:一条数据总线用于传送数据(操作数)进出 CPU;一条地址总线用于访问数据;一条总线用于传送操作码;一条地址总线用于访问操作码。分离的代码和数据总线就是常说的哈佛结构。上一节中讨论过 PIC18 的哈佛结构。

6. 特性 6

因为 CISC 有庞大的指令系统,而每一条指令又有多个不同的寻址方式,微指令(微代码)就是用来实现它们的。对于多数的 CISC 处理器来说,CPU 内微指令的实现会占用超过 60%的晶体管。然而,对于 RISC 来说,由于指令很少,所以它们的实现都是采取硬连接方式。RISC 指令的硬件实现所使用的晶体管不超过 10%。

7. 特性 7

RISC 使用读取/存储结构。对于 CISC 微处理器,数据可以在内存里操作。例如,对于指令 `ADD Reg, Memory`,微处理器必须把外部存储器的内容先传送到 CPU,然后将它同寄存器的内容相加,最后把所得的结果传回至外部的存储器地址。问题是,在从外部存储器读入数据时可能会出现延时。于是,指令执行的整个过程将会停止,这将阻止流水线上的其他指令的执行。在 RISC 处理器中,设计人员摒弃了这类指令。在 RISC 里,指令只能从外部存储器读取数据到寄存器或者是把寄存器的内容存放到外部存储器。在寄存器和外部存储器的内容之间没有直接实现算术运算和逻辑运算操作的方法。所有这些指令都需要首先把操作数传送至寄存器和 CPU,然后执行算术运算或者逻辑运算操作,再把结果送回到存储器。这种设计理念在 1976 年由 Cray 1 超级计算机首先实现,后来被广泛称作读取/存储结构。在上一节中,我们注意到了文件寄存器(内存)和 WREG 寄存器之间的算术运算和逻辑运算操作,但是

86 并没涉及 ROM 地址和文件寄存器地址的操作。例如，PIC18 里面不存在 ADDW ROM-Loc 指令。

在结束讨论 RISC 处理器之际，读者会发现一件趣事：尽管 RISC 技术是由 IBM 的科学家在 20 世纪 70 年代中期开发的，可是 RISC 的优点却是由加州大学伯克利分校的 David Patterson 在 1980 年总结归纳的，并引起计算机科学家的注意。值得指出的是，在最近几年里，CISC 处理器(如 Pentium)在设计上已吸收了一些 RISC 的优点。这是增强 x86 处理器的处理能力和保持竞争力的唯一出路。当然，使用大量的晶体管实现是必不可少的，因为 8086/286/386 处理器的所有 CISC 指令和 DOS 的遗留软件都需要处理。

2.9.3　复习题

1. RISC 和 CISC 分别代表什么？
2. 判断对错：大多数的 CISC 指令执行都要占用 2、3 或者更多时钟周期，而 RISC 只需要 1 个时钟周期。
3. RISC 处理器通常有________(很多，很少)的通用寄存器。
4. 判断对错：在 RISC 处理器(如 PIC18)中，不存在形如 ADD WREG, ROMmemory 的指令。
5. PIC18 有多少条指令？它是不是 RISC？
6. 判断对错：CISC 有各种长度的指令，而 RISC 的指令都是长度相同的。
7. 对于 RISC 的 ADD 指令，下面的哪个操作不会出现？
   (a) 寄存器到寄存器　(b) 立即数到寄存器　(c) 内存到内存
8. 判断对错：哈佛结构使用相同的地址和数据总线去读取代码和数据。

2.10　使用 MPLAB 仿真器查看寄存器和存储器

PIC 微控制器有很多优秀的开发工具和支持系统，大多数都是免费或低廉的。MPLAB 是由 Microchip 公司免费提供的集编译器、连接器、仿真器于一体的开发工具，可以从 www.microchip.com 网站免费下载。关于如何使用 MPLAB 的编译器和仿真器，请登录网站 http://www.MicroDigitalEd.com 查询详情。

很多汇编器和 C 编译器都带有仿真器。仿真器允许设计人员在执行每条指令(单步执行)后可以查看寄存器和存储器的内容。强烈建议读者使用仿真器来单步执行本章和以后章节中的程序。使用仿真器单步执行程序，除了可以发现程序中的语法错误外，还能让读者对 87 微控制器结构有更为深刻的理解和认识。图 2-15～图 2-17 是来自 MPLAB 的 PIC 仿真器的截图。

关于使用 MPLAB 的资料手册请登录下面的网址：http://www.MicroDigitalEd.com

小结

本章首先介绍了 PIC 的主要寄存器，包括 WREG、SFR、通用数据 RAM 和程序计数器。这些寄存器的用法都以程序例子的形式给予了详细的说明。汇编语言程序的创建过程，可以总结为编写源文件、汇编、连接和执行。PC(程序计数器)寄存器总是指向等待执行的下一条指令。本章还介绍了 PIC 使用程序 ROM 空间的方法，因为 PIC 汇编程序的程序员必须知道程序放在 ROM 的哪些位置，以及还有多少内存是可用的。

Special Function Registers

Address	SFR Name	Hex	Decimal	Binary	Char
0F80	PORTA	00	0	00000000	.
0F81	PORTB	00	0	00000000	.
0F82	PORTC	00	0	00000000	.
0F83	PORTD	00	0	00000000	.
0F84	PORTE	00	0	00000000	.
0F89	LATA	00	0	00000000	.
0F8A	LATB	00	0	00000000	.
0F8B	LATC	00	0	00000000	.
0F8C	LATD	00	0	00000000	.
0F8D	LATE	00	0	00000000	.
0F92	TRISA	00	0	00000000	.
0F93	TRISB	00	0	00000000	.
0F94	TRISC	00	0	00000000	.
0F95	TRISD	00	0	00000000	.
0F96	TRISE	00	0	00000000	.
0F9D	PIE1	00	0	00000000	.
0F9E	PIR1	00	0	00000000	.
0F9F	IPR1	00	0	00000000	.
0FA0	PIE2	00	0	00000000	.
0FA1	PIR2	00	0	00000000	.
0FA2	IPR2	00	0	00000000	.

图 2-15　MPLAB 仿真器的 SFR 窗口

File Registers

Address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
0000	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0010	9E	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

Hex　Symbolic

图 2-16　MPLAB 仿真器的文件寄存器(数据 RAM)窗口

Program Memory:1

Line	Address	Opcode	Disassembly
1	0000	0E0A	MOVLW 0xa
2	0002	6E25	MOVWF 0x25, ACCESS
3	0004	0E00	MOVLW 0
4	0006	0F03	ADDLW 0x3
5	0008	0625	DECF 0x25, F, ACCESS
6	000A	E1FD	BNZ 0x6
7	000C	6E81	MOVWF 0xf81, ACCESS

Opcode Hex　Machine　Symbolic

图 2-17　MPLAB 仿真器的程序(代码)ROM 窗口

一系列由指令或者伪代码(也称作命令)构成的语句组成汇编语言程序。指令由汇编器翻译成机器代码。伪指令不会被翻译成机器代码,它们用于指示汇编器如何把指令翻译成机器代码。有些用于定义数据的伪指令被称为*数据伪指令*。数据以字节形式递增存放。数据可以用二进制、十六进制、十进制或者ASCII形式来表示。

标志位对于程序员来说是很有用的,因为它们代表着一定的条件,例如由指令执行引起的进位标志或者归零标志。本章还介绍了RISC和哈佛结构的概念。

88～89

RISC结构使得微控制器的处理能力更加强大。它采用简单的指令集和大量的寄存器。哈佛结构使得CPU读取代码和数据的速度更加快速。PIC18更宽的数据总线使得在每个时钟周期内就能读取一条指令,因为典型的PIC指令是2 B长的。

习题

1. PIC18是________位的微控制器。
2. WREG寄存器有________位宽。
3. MOVLW中的立即数是________位宽。
4. 可以存放到WREG的最大的十六进制数是________。
5. 要把WREG赋值为65H,指令 MOVLW # 65H 中的#号是________(不需要、可选、必要)的。
6. 下面指令的结果是多少?结果会放在什么地方?
```
MOVLW  15H
ADDLW  13H
```
7. 下面哪些指令是非法的?
(a) MOVLW 500　(b) MOVLW 50　(c) MOVLW 00
(d) MOVLW 255H　(e) MOVLW 25H　(f) MOVLW F5H
(g) MOVLW mybyte,50H
8. 下面哪些指令是非法的?
(a) ADDLW 300H　(b) ADDLW 50H　(c) ADDLW $500
(d) ADDLW 255H　(e) ADDLW 12H　(f) ADDLW 0F5H
(g) ADDWL 25H
9. 下面指令的结果是多少?会放在什么地方?
```
MOVLW  25H
ADDLW  1FH
```
10. 下面指令的结果是多少?会放在什么地方?
```
MOVLW  15H
ADDLW  0EAH
```
11. 指令 ADDWL K 中K允许的最大十六进制数是________。
12. 判断对错:在PIC18里有很多个WREG寄存器。
13. PIC的数据RAM由________(EEPROM,SRAM)构成。
14. 判断对错:PIC的数据RAM也被称作文件寄存器。
15. 判断对错:SFR是文件寄存器空间的一部分。
16. 判断对错:通用RAM不是文件寄存器空间的一部分。
17. 判断对错:PIC18系列所有芯片的文件寄存器大小都相等。

18. 如果我们把 SFR 和通用 RAM 的空间加在一起，会得到________的总存储空间。
19. 找出下面 PIC 芯片的文件寄存器大小。
    (a)PIC12508　　(b)PIC16F84　　(c)PIC18F8772
20. PIC18 的 EEPROM 和数据 RAM 的区别是什么？
21. 能有不带 EEPROM 的 PIC 芯片吗？
22. 能有不带文件寄存器的 PIC 芯片吗？
23. 访问存储器有________字节的空间。
24. 画出访问存储器中 SFR 和 GP RAM 的地址分布。
25. PIC18 最大存储空间是多少？
26. PIC18 用于文件寄存器的最大存储空间是多少？
27. 访问存储区里临时寄存器的地址范围是多少？
28. 给出把 30H 和 97H 放入地址 5 和 6 的简单代码。
29. 给出把 55H 放入地址 0～8 的简单代码。
30. 给出把 5FH 放入 SFR 的 PortB 的简单代码。
31. 判断对错：不能直接把立即数送入临时区。
32. 判断对错：指令 ADDWF fileReg, D 涉及文件寄存器和 WREG。
33. 在第 32 题里，要把结果放入 WREG，位 D 应为________。
34. 在第 32 题里，要把结果放入 fileReg，位 D 应为________。
35. 写出简单的代码完成下面的操作：(a) 把 11H 放入地址 0～5；(b)把所有值相加并在过程中把结果送入 WREG。
36. 重复第 35 题，但最后结果放入地址 5。
37. 写出简单的代码完成下面的操作：(a) 把 15H 送入地址 7；(b)把它加到 WREG，重复 5 次，并把结果放在 WREG。在加法执行前，WREG 应为 0。
38. 重复第 37 题，但结果放在地址 7。
39. 指令 MOVWF 和 MOVF 的区别是什么？
40. 编写简单的代码，使地址 8 的内容取反，并把结果放入 WREG。
41. 判断对错：可以使用 MOVFF 指令把数据在文件寄存器范围内随意复制。
42. 编写简单的代码，把地址 8 的数据复制到 PORTC(a)使用 WREG；(b)不使用 WREG。
43. 状态寄存器是一个________位的寄存器。
44. 状态寄存器的哪些位分别是用作 C 和 DC 状态标志位的？
45. 状态寄存器的哪些位分别是用作 O 和 NV 状态标志位的？
46. 使用 ADDLW 指令，C 位什么时候增加？
47. 使用 ADDLW 指令，DC 位什么时候增加？
48. 下面的程序执行完后，状态标志位 C 和 Z 是多少？

```
MOVLW   FFH
ADDLW   1
```

49. 下面的程序执行完后，状态标志位 C 是多少？

```
(a) MOVLW   54H
    ADDLW   0C4H
(b) MOVLW   00
    ADDLW   FFH
```

90
91

(c) MOVLW FFH
ADDLW 05H

50. 编写简单的程序，执行55H的加法运算5次。
51. 列出下面每条数据的十六进制值。

```
MYDAT_1    EQU  55
MYDAT_2    EQU  D'98'
MYDAT_3    EQU  A'G'
MYDAT_4    EQU  0x50
MYDAT_5    EQU  D'200'
MYDAT_6    EQU  A'A'
MYDAT_7    EQU  AAH
MYDAT_8    EQU  D'255'
MYDAT_9    EQU  B'10010000'
MYDAT_10   EQU  B'01111110'
MYDAT_11   EQU  D'10'
MYDAT_12   EQU  D'15'
```

52. 列出下面每条数据的十六进制值。

```
DAT_1      EQU  22
DAT_2      EQU  56H
DAT_3      EQU  B'10011001'
DAT_4      EQU  D'32'
DAT_5      EQU  0xF6
```

53. 写出简单的代码完成下面的操作：(a)11H送给地址0～5；(b)将所有值相加并把结果放入WREG。使用EQU来分配名字R0～R5给地址0～5。
54. 汇编语言是________(低级，高级)语言，而C语言是________(低级，高级)语言。
55. C语言和汇编语言相比，哪个的代码生成效率(即ROM空间的使用率)更高？
56. 什么程序会生成o(目标)文件？
57. 判断对错：源文件的扩展名是asm。
58. 什么文件提供错误信息的列表？
59. 判断对错：源文件可以是非ASCII码文件。
60. 判断对错：每个源文件都有ORG和END指令。
61. ORG和END指令会产生操作码吗？
62. 为什么ORG和END指令又称作伪指令？
63. 判断对错：ORG和END指令会在列表文件中出现。
64. 判断对错：连接器生成的文件带有asm的扩展名。
65. 判断对错：连接器生成的文件带有hex的扩展名。
66. 带有________扩展名的文件可以被下载到PIC的ROM。
67. 给出MPLAB仿真器生成的3个生成文件的扩展名。
68. PIC18系列的芯片通电后都是从________地址开始运行的。
69. 一个程序员把起始操作码放在了地址100H。那么在微控制器通电后会发生什么情况呢？
70. 试确定下面各条指令所占用的字节长度。

(a) MOVLW 5H　　(b) MOVLW 9FH　　(c) ADDLW 50H
(d) ADDLW 0　　(e) MOVLM 0x41　　(f) MOVLW 20

92

(g) ADDLW d'200' (h) GOTO

71. 编写程序实现以下功能:(a)把 5 个数字依次存放到从 0 开始的 RAM 地址;(b)把每个数加到 WREG 中,并把结果存放在 RAM 的地址 6;(c)使用程序列表文件,查看 ROM 的存储地址和存储内容。
72. 使用你选择的程序列表文件,查看 ROM 地址及其内容。
73. 试确定下面片上程序 ROM 的最后一位地址。
    (a)48KB 的 PIC (b)96KB 的 PIC (c)64KB 的 PIC
    (d)16KB 的 PIC (e)128KB 的 PIC
74. 请给出 PIC18 程序计数器的最大值和最小值(用十六进制表示)。
75. 假设某 PIC 芯片的片上 ROM 最后的地址是 7FFFH,那么它的片上 ROM 大小为多少?
76. 若片上 ROM 的最后一个地址是 3FFH,请重算第 75 题。
77. 试确定下列地址范围所对应的 PIC18 芯片片上程序 ROM 的大小(用 KB 表示)。
    (a) 0000~1FFF (b) 0000~3FFF (c) 0000~5FFF
    (d) 0000~BFFF (e) 0000~FFFF (f) 00000~1FFFF
    (g) 00000~2FFFF (h) 00000~3FFFF
78. 试确定下列地址范围所对应的 PIC18 芯片片上程序 ROM 的大小(用 KB 表示)。
    (a) 00000~4FFFF (b) 00000~3FFFF (c) 00000~5FFFF
    (d) 00000~7FFFF (e) 00000~BFFFF (f) 00000~FFFFF
    (g) 00000~17FFFF (h) 00000~1FFFFF
    备注:上面的地址范围有可能还没有生产。
79. PIC18 芯片的 ROM 数据宽度是多少?
80. 在 PIC18 里,CPU 和程序 ROM 的数据总线宽度是多少?
81. 参阅图 2-11,确定存储空间 4K×16 的奇地址和偶地址。
82. 在第 81 题里,ROM 空间的大小是多少?(用 KB 表示。)
83. 参阅图 2-11,确定存储空间 16K×16 的奇地址和偶地址。
84. 在第 83 题里,ROM 空间的大小是多少?(用 KB 表示。)
85. 解释哈佛结构以及它是如何更快速地处理代码和数据的。
86. 在外部存储器到 CPU 之间采用哈佛结构的障碍在哪里?
87. 试解释指令 MOVLW K 里的 K 为什么不能大于十进制数 255。
88. 试解释指令 ADDLW K 里的 K 为什么不能大于十进制数 255。
89. 试指出指令 MOVWF fileReg 的指令长度,并解释它为什么能覆盖 PIC18 文件寄存器的全部范围。
90. 试指出指令 MOMVFF source,dest 的指令长度,并解释它为什么能覆盖 PIC18 文件寄存器的全部范围。
91. 试解释在指令 GOTO target_addr 中程序计数器的最低有效位为 0 的原因。
92. 试解释指令 GOTO target_addr 不会跳转到奇地址。
93. 对于第 92 题,解释其原因。
94. 试解释指令 GOTO target_addr 是如何能覆盖 PIC18 的整个 2MB 的地址空间的。
95. RISC 和 CISC 分别代表什么?
96. 在________(RISC,CISC)结构中存在 1B、2B、3B 或者 4B 长的指令。
97. 在________(RISC,CISC)结构中,指令的长度是固定的。

98. 在________(RISC,CISC)结构中,多数的指令在一个或两个时钟周期里执行完毕。
99. 在________(RISC,CISC)结构中,可以使用一条指令完成寄存器和外部存储器的加法运算。
100. 判断对错:CISC里的大多数指令在一个或两个时钟周期里执行完毕。

复习题答案

2.1 节

1. MOVLW 0x34
2. MOVLW 0x16 ADDLW 0xCD
3. 错误
4. 十六进制 FF 和十进制 255
5. 8

2.2 节

1. 正确。 2. 文件寄存器 3. 正确。 4. 正确。 5. 8 6. 256 7. 4096

2.3 节

1. 正确。
2. MOVLW 0x16 MOVWF 0 MOVLW 0xFD ADDWF 0,F
3. 正确。 4. FF,255 5. WREG

2.4 节

1. 状态寄存器 2. 8位
3. D5、D6、D7
4.

十六进制	二进制
9F	1001 1111
+ 61	+ 0110 0001
100	10000 0000

由此可得 C=1,DC=1,Z=1

5.

十六进制	二进制
82	1000 0010
+22	+ 0010 0010
A4	1010 0100

由此可得 C=0,DC=0,Z=0

6.

十六进制	二进制
67	0110 0111
+99	+ 1001 1001
100	10000 0000

由此可得 C=1,DC=1,Z=1

2.5 节

1.
```
DATA1 EQU 9FH
DATA2 EQU 0x9F
DATA3 EQU H'9F'
```
2.
```
DATA1 EQU 99H
DATA2 EQU D'99'
```

DATA3 EQU B'10011001'

3. 如果要改变一个值,不是在它出现的每个地方都执行修改,只需要在一个地方修改一次即可。
4. (a) 34H (b) 1FH 5. WREG=15H
6. 地址值 0x20=(0X95)
7. 0CH+10H=1CH 将被存放在到文件寄存器的地址 63H。

95

2.6 节

1. 实际的操作由诸如 MOV 和 ADD 等指令来完成。伪指令,也叫汇编命令,用于指导汇编器工作。
2. 指令助记符,伪指令 3. 错误。 4. 除(c)之外的全部
5. 汇编命令 6. 正确。 7. (c)

2.7 节

1. 正确。 2. 正确。 3. (a) 4. (b)到(e) 5. (d)和(e)

2.8 节

1. 21 2. 正确。 3. 0000H 4. 2
5. 带有 32 KB,共计 32 768(32×1024=32 768)B,ROM 地址空间从 0000 到 7FFFH。
6. 4 7. 正确。

2.9 节

1. CISC 表示复杂指令集计算机,RISC 表示精简指令集计算机。
2. 正确。 3. 小 4. 正确。 5. 75,是 6. 正确。 7. (c) 8. 错误。

96

第3章
分支、调用和时延循环

学习目标：

- ❑ PIC 汇编语言循环指令程序的编写
- ❑ PIC 汇编语言条件分支指令的编写
- ❑ 决定条件分支指令的条件
- ❑ 使用 GOTO(长跳转)实现无条件转移的指令编写
- ❑ 条件分支指令的目标地址的计算
- ❑ PIC 子例程的编写
- ❑ 栈及其在子例程中的应用
- ❑ PIC 中的流水线技术
- ❑ PIC 中晶振频率和指令周期的关系
- ❑ PIC 时延程序的编写

在指令执行过程中，常常需要把程序控制转移到不同的位置。在 PIC 中有许多指令可以实现这种功能。本章将介绍 PIC 汇编语言中所有的控制转移指令。3.1 节将讨论循环指令以及用于实现条件转移和无条件分支(跳转)的指令。3.2 节将介绍栈和调用指令。3.3 节将讨论 PIC18 的流水线技术，以及指令时序和时延子例程。

3.1 分支指令和循环

本节将首先讨论在 PIC 中是如何实现循环操作的，然后介绍分支(跳转)指令，包括条件转移指令与无条件转移指令。

3.1.1 PIC 的循环语句

重复地执行一段指令或者一个操作若干次，就称为循环。循环是广泛使用的编程方法之一。在 PIC 里，有几种实现重复操作的方法。一种方法是重复地执行运算操作，直到结束，如下所示。

```
MOVLW 0       ;WREG = 0
ADDLW 3       ;add value 3 to WREG
ADDLW 3       ;add value 3 to WREG(W = 6)
ADDLW 3       ;add value 3 to WREG(W = 9)
ADDLW 3       ;add value 3 to WREG(W = 0Ch)
ADDLW 3       ;add value 3 to WREG(W = 0Fh)
```

在上面的程序里，把数值 3 加到 WREG 里，执行了 5 次，即有结果：5×3=15=0Fh。上面的程序存在这样一个问题：如果重复执行的次数是 50 或者 100，那么程序所需要的代码空间就会很大。一种更好的解决办法是使用循环。PIC 提供了两种实现循环的方法，下面将分别介绍。

1. DECFSZ 指令与循环

DECFSZ(递减文件寄存器跳零)是广泛使用的指令，支持从 PIC12 到 PIC18 的所有 PIC 微控制器。其指令格式为：

```
DECFSZ fileReg,d;decrement fileReg and skip next instruction if 0
```

在这条指令里，fileReg 自减 1。当 fileReg 的内容为 0 时，程序计数器将跳过下一条指令。如果把指令 GOTO target 紧接着放置在这条指令的下面，就可以实现循环功能。指令 GOTO target 的目标地址就是循环的起点，如例 3-1 和例 3-2 所示。图 3-1 给出了 DECFSZ 指令的流程图。要熟悉其中的图形符号，可参阅附录 D 的流程图结构。流程图是一种形象描述程序执行顺序的常用方法。在设计程序时，强烈建议读者使用流程图。

98

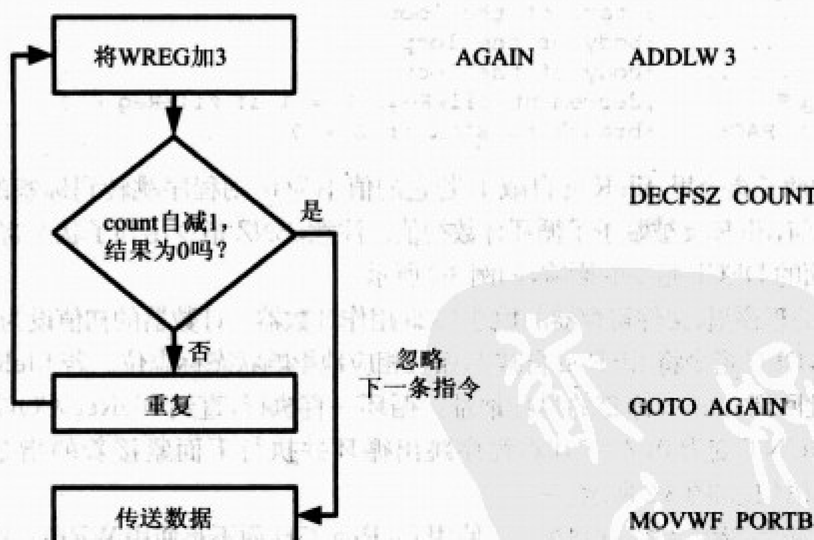


图 3-1 DECFSZ 指令的流程图

例 3-1 编写一个程序实现：(a)将 WREG 清零；(b)将 WREG 加 3，重复执行 10 次，所得结果放入 PORTB 的 SFR。请使用 DECFSZ 指令实现循环。

解：

```
;this program adds value 3 to WREG ten times

COUNT   EQU 0x25            ;use loc 25H for counter
        MOVLW   d'10'       ;WREG = 10 (decimal) for counter
        MOVWF   COUNT       ;load the counter
        MOVLW   0           ;WREG = 0
```

```
AGAIN   ADDLW   3           ;add 03 to WREG (WREG = sum)
        DECFSZ  COUNT,F     ;decrement counter, skip if count = 0
        GOTO    AGAIN       ;repeat until count becomes 0
        MOVWF   PORTB       ;send sum to PORTB SFR
```

值得注意的是,DECFSZ 指令会将初值为 10 的计数器(fileReg loc 0x25)自减 1,然后计数器的值变为 9。因为它不是 0,所以 CPU 将执行 GOTO AGAIN 指令。执行 GOTO AGAIN 返回循环体的起点。接下来,计数器再自减 1,其值变为 8。由于结果不是 0,所以再次执行 GOTO 操作。程序就像这样一直运行,直到计数器的值变为 0。当计数器的值变为 0 时,程序跳过 GOTO 指令,从而跳出了循环,执行 MOVWF PORTB 指令。注意,这里使用的是语句 DECFSZ COUNT, F,而不是 DECFSZ COUNT, W,其原因是设计人员希望计数值在下一次迭代里继续变化。如果使用语句 DECFSZ COUNT, W,程序将永远跳不出循环,因为 COUNT 始终等于 9,而且自减后的值放回到了 WREG 寄存器。

99

2. 使用 BNZ 指令实现循环

BNZ(Branch if Not Zero,非 0 则跳转)指令适用于 PIC18 系列,而不适用于早期的 PIC 系列(如 PIC16 和 PIC12)。它的执行用到了状态寄存器的零标志位。BNZ 指令的用法如下:

```
BACK   .........    ;start of the loop
       .........    ;body of the loop
       .........    ;body of the loop
       DECF         ;decrement fileReg, Z = 1 if fileReg = 0
       BNZ  BACK    ;branch to BACK if Z = 0
```

在最后的两条指令里,fileReg 自减 1;若它的值不为 0,则程序跳转到标签的目标地址。在循环开始之前,fileReg 被赋予了循环计数初值。注意,BNZ 指令用到了状态寄存器的 Z 标志位,它受前面的 DECF 指令的影响,如例 3-2 所示。

在例 3-2 的程序里,文件寄存器的地址 0x25 用作计数器。计数器的初值设为 10。在每次迭代(循环)里,DEC 指令将 fileReg 自减 1,而且相应地影响状态标志位。若 fileReg 不为 0(Z≠0),则程序跳回到 AGAIN 标签的目标地址。循环一直执行,直到 fileReg COUNT 变为 0。当 fileReg COUNT 变为 0(Z=0)时,程序跳出循环并执行下面紧接着的指令,在这里是 MOVWF PORTB 语句,如图 3-2 所示。

值得注意的是,指令 DECF COUNT, F 使用 fileReg 25H 而不是使用 WREG 来作为计数值的寄存器。如果用 WREG 作为 DECF 指令的目标地址,那么程序将会陷入死循环,因为 COUNT 会一直保持初值为 10。

例 3-2 编写程序实现:(a) 将 WREG 清零;(b) 将 WREG 加 3,执行 10 次。请使用零标志位和 BNZ 指令编写程序。

解:

```
;this program adds value 3 to the WREG ten times

     COUNT EQU 0x25       ;use loc 25H for counter

     MOVLW d'10'          ;WREG = 10 (decimal) for counter
     MOVWF COUNT          ;load the counter
```

```
        MOVLW  0              ;WREG = 0
AGAIN   ADDLW  3              ;add 03 to WREG (WREG = sum)
        DECF   COUNT, F       ;decrement counter
        BNZ    AGAIN          ;repeat until COUNT = 0
        MOVWF  PORTB          ;send sum to PORTB SFR
```

100

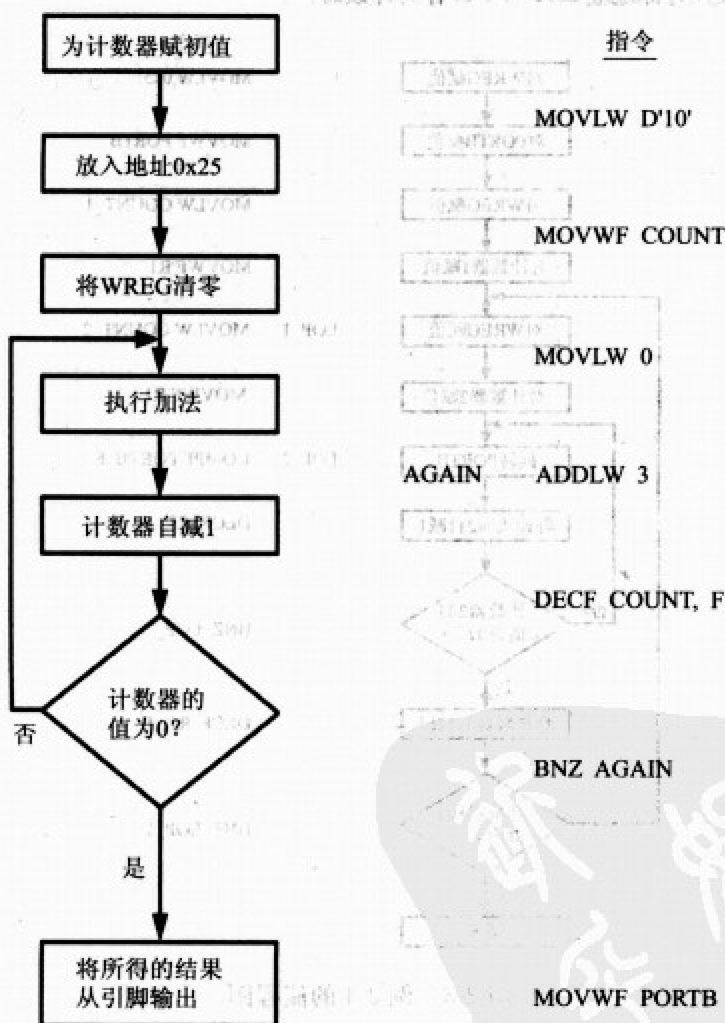


图 3-2 例 3-2 的流程图

例 3-3 在例 3-2 中，试确定可供循环执行的最大次数。

解：因为 fileReg 的 COUNT 是 8 位寄存器，可存放的最大数值是 FFH（十进制数为 255）。因此，程序最多可执行 255 次循环。若要突破这个限制，请参阅例 3-4。

101

3.1.2 循环嵌套

正如例 3-2 所示，循环的最大计数值是 255。那如何才能实现循环次数大于 255 呢？要达到这个目的，可以把一个循环放在另一个循环里，这称为*嵌套循环*。嵌套循环使用两个寄存

器来存放计数初值。请看例 3-4。

例 3-4　编写程序：(a) 将 PORTB SFR 寄存器赋值为 55H；(b) 将 PORT B 取反 700 次。

解：因为 700 大于 255(所有寄存器存放的最大数值)，所以使用两个寄存器来存放计数值。下面给出的代码将说明如何使用存储地址 25H 和 26H 作为计数器。

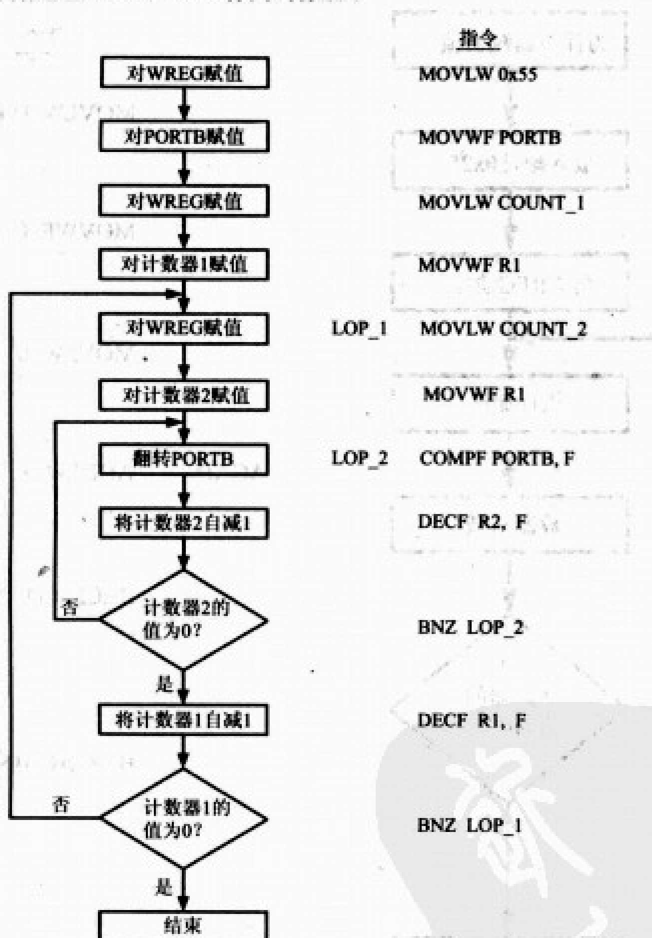


图 3-3　例 3-4 的流程图

```
        R1 EQU 0x25
        R2 EQU 0x26
        COUNT_1 EQU d'10'
        COUNT_2 EQU d'70'
        MOVLW 0x55          ;WREG = 55h
        MOVWF PORTB         ;PORTB = 55h
        MOVLW COUNT_1       ;WREG = 10, outer loop count value
        MOVWF R1            ;load 10 into loc 25H (outer loop count)
LOP_1   MOVLW COUNT_2       ;WREG = 70, inner loop count value
        MOVWF R2            ;load 70 into loc 26H
LOP_2   COMPF PORTB, F      ;complement Port B SFR
        DECF R2, F          ;dec fileReg loc 26 (inner loop)
        BNZ LOP_2           ;repeat it 70 times
        DECF R1, F          ;dec fileReg loc 25 (outer loop)
        BNZ LOP_1           ;repeat it 10 times
```

在这个程序中，fileReg 地址 0x26 用来存放内层循环计数值。对于指令 BNZ LOP_2，无论地址 26H 是否为 0，程序将结束循环并执行 DECF R1, F。如果计数值不为 0，那么该指令将强制 CPU 加载内层循环计数初值 70，于是，内层循环重新开始。这个过程将一直持续到地址 25 的内容变为 0，此时，外层循环结束。

存储地址	数值	
25	10	R1
26	70	R2

102

3.1.3 循环 100 000 次

使用 2 个寄存器可得到的最大循环次数为 65 025(255×255=65 025)，若使用 3 个寄存器则可以实现 1600(2^{24})多万次循环。下面给出实现 100 000 次循环的指令代码：

```
R1 EQU 0x1                    ;assign RAM loc for the R1-R2
R2 EQU 0x2
R2 EQU 0x3

COUNT_1 EQU D'100'            ;fixed value for 100,000 times
COUNT_2 EQU D'100'
COUNT_3 EQU D'10'

        MOVLW  0x55
        MOVWF  PORTB
        MOVLW  COUNT_3
        MOVWF  R3
LOP_3   MOVLW  COUNT_2
        MOVWF  R2
LOP_2   MOVLW  COUNT_1
        MOVWF  R1
LOP_1   COMPF  PORTB, F
        DECF   R1  ,F
        BNZ    LOP_1
        DECF   R2,  F
        BNZ    LOP_2
        DECF   R3,  F
        BNZ    LOP_3
```

3.1.4 其他的条件转移指令

表 3-1 简要地罗列了 PIC 的条件分支指令。关于条件分支指令的更详细介绍，请参阅附录 A。注意，有些指令只有在满足一定条件时才跳转，如 BZ 指令(当 Z=1 时，执行跳转)和 BC 指令(当 C=1 时，执行跳转)。接下来，引入一些例子来说明条件跳转指令。

表 3-1 PIC 条件分支(跳转)指令

指 令	结 果
BC	C=1 时跳转
BNC	C≠0 时跳转
BZ	Z=1 时跳转
BNZ	Z≠0 时跳转
BN	N=1 时跳转
BNN	N≠0 时跳转
BOV	OV=1 时跳转
BNOV	OV≠0 时跳转

1. BZ 指令(若 Z=1 则跳转)

该指令将检查 Z 标志位。若 Z=1，则程序

会转到目标地址。请看下面的例子[①]：

```
OVER  MOVF  PORTB,W        ;read Port B and put it in WREG
      JZ  OVER             ;jump if WREG is zero
```

在这个程序里，如果 PORTB 为 0，程序跳到 OVER 标识的地方。程序将一直循环，直到 PORTB 的值不为 0。注意，BZ 指令可以用来判断任何 fileReg 或者 WREG 寄存器是否为 0。更重要的是，在使用 BZ 指令时，不需要执行算术指令(如减 1 运算)。请看例3-5。

103~104

例 3-5 试编写程序，查看 fileReg 地址 0x30 的值是否为 0。如果是 0，请将数值 55H 赋值给 fileReg 地址 0x30。

解：

```
      MYLOC EQU 0x30
      MOVF  MYLOC,F           ;copy MYLOC to itself
      BNZ   NEXT              ;branch if MYLOC is not zero
      MOVLW 0x55
      MOVWF MYLOC             ;put 0x55 if MYLOC has zero value
NEXT  ...
```

2. BNC 指令(若没有进位，则跳转，即在 CY=0 时执行跳转)

该指令判断状态寄存器的进位标志位来决定是否跳转。在执行 BNC label 时，处理器会检查进位标志位是否为 1(C=1)。若标志位不是 1，则 CPU 将从标签指向的地址上读取指令并执行。若标志位 C=1，则程序不会发生跳转，并且执行紧接 BNC 的下一条指令。例 3-6 讲述了怎样使用 BNC 指令来执行结果大于 FFH 的加法。还要注意的一条指令是 BC label。在 BC 指令中，若 C=1，则跳转到目标地址。关于这些指令的更多应用例子，请参阅第 5 章。

表 3-1 中的其他条件转移指令将在第 5 章讨论有符号数的算术运算操作时再加以介绍。

例 3-6 计算数值 79H、F5H 和 E2H 的和。然后把结果的低字节和高字节分别放入 fileReg 的地址 5 和 6。

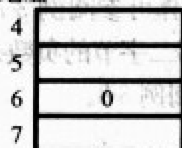
解：

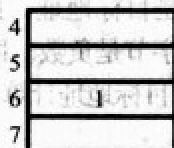
```
L_Byte EQU 0x5             ;assign RAM loc 5 to L_byte of sum
H_Byte EQU 0x6             ;assign RAM loc 6 to H_byte of sum

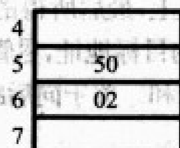
       ORG 0h
       MOVLW 0x0           ;clear WREG(WREG = 0)
       MOVWF H_Byte        ;H_Byte = 0
       ADDLW 0x79          ;WREG = 0 + 79H = 79H, C = 0
       BNC   N_1           ;if C = 0, add next number
       INCF  H_Byte,F      ;C = 1, increment (now H_Byte = 0)
N_1    ADDLW 0xF5          ;WREG = 79 + F5 = 6E and C = 1
       BNC   N_2           ;branch if CY = 0
       INCF  H_Byte,F      ;C = 1, increment (now H_Byte = 1)
N_2    ADDLW 0xE2          ;WREG = 6E + E2 = 50 and C = 1
       BNC   OVER          ;branch if C = 0
```

① 此例中的 JZ 应为 BZ，原书有误。——译者注

```
        INCF   H_Byte,F     ;C = 1, increment (now H_Byte = 2)
OVER    MOVWF  L_Byte       ;now L_Byte = 50H, and H_Byte = 02
        END
```







105

3.1.5 所有的条件分支指令都是短跳转

必须指出的是，所有的条件跳转指令都是短跳转的，换而言之，目标地址必须在程序计数器(PC)的256B的范围内。这个概念将在后面继续讨论。

例3-7 下面是例3-6的列表文件，请验证向前跳转地址的计算。

```
行        程度计数器       操作码                 助记符
LOC     OBJECT LINE   SOURCE TEXT
        CODE
  VALUE

  00000005      00001 L_Byte EQU 0x5        ;assign RAM Loc 5 to L_byte of sum
  00000006      00002 H_Byte EQU 0x6        ;assign RAM Loc 6 to H_byte of sum
                00003
000000          00004        ORG 0h
000000 0E00     00005        MOVLW  0x0     ;clear WREG(WREG=0)
000002 6E06     00006        MOVWF  H_Byte  ;H_Byte = 0
000004 0F79     00007        ADDLW  0x79    ;WREG = 0 + 79H = 79H, C = 0
000006 E301     00008        BNC    N_1     ;if C = 0, add next number
000008 2A06     00009        INCF   H_Byte,F ;C = 1, increment (now H_Byte = 0)
00000A 0FF5     00010 N_1    ADDLW  0xF5    ;WREG = 79 + F5 = 6E and C = 1
00000C E301     00011        BNC    N_2     ;branch if CY = 0
00000E 2A06     00012        INCF   H_Byte,F ;C = 1, increment (now H_Byte = 1)
000010 0FE2     00013 N_2    ADDLW  0xE2    ;WREG = 6E + E2 = 50 and C = 1
000012 E301     00014        BNC    OVER    ;branch if C = 0
000014 2A06     00015        INCF   H_Byte,F ;C = 1, increment (now H_Byte = 2)
000016 6E05     00016 OVER   MOVWF  L_Byte  ;now L_Byte = 50H, and H_Byte = 02
                00017        END
```

解：

首先，要注意的是，BNC指令是向前跳转的。向前跳转的目标地址等于下一条指令的PC值与跳转指令的第二个字节的内容乘以2的代数和。因为每条指令都是2字节大小的。在第6行，指令BNC N_1的操作码E3和操作数01分别位于地址000006和000007。01×02=02是一个相对地址，它相对于下一条指令INCF(地址为000008)。把000002和000008相加，就得到标签N_1的目标地址，即为00000A。同样，000011行的指令BNC N_2和000014行的指令BNC OVER都是向前跳转的，因为相对地址均大于0。

106

3.1.6 短转移地址的计算

所有的条件分支指令(如BNC、BZ和BNZ)都是短转移的，因为它们都是2B长的指令。

这些指令的第一个字节是操作码,第二个字节是相对地址。目标地址是相对于程序计数器的值。若第二个字节是正数,则向前跳转。相反,若第二个字节是负数,则向后跳转。第二个字节可以是从−127到+128范围内的任意值。将指令的第二个字节乘以2所得的乘积,再加到下一条指令的PC值上,最后所得的和就是目标地址。具体计算可参阅例3-7。类似地,可以计算出向后跳转时的目标地址,尽管第二字节是负数。即,把第二字节的负数乘以2后再同下一条指令的PC值求和。关于向后跳转的目标地址计算,请参阅例3-8。

例3-8 下面是例3-2的列表文件,请验证向后跳转地址的计算。

解:

```
LOC    OBJECT LINE   SOURCE TEXT
       CODE
  VALUE

  00000025    00001  COUNT  EQU 0x25      ;use loc 25H for counter
000000        00002         ORG 0h
000000 0E0A   00003         MOVLW d'10'    ;WREG = 10 (decimal) for counter
000002 6E25   00004         MOVWF COUNT    ;load the counter
000004 0E00   00005         MOVLW 0        ;WREG = 0
000006 0F03   00006  AGAIN  ADDLW 3        ;add 03 to WREG (WREG = sum)
000008 0625   00007         DECF COUNT, F  ;decrement counter
00000A E1FD   00008         BNZ AGAIN      ;repeat until COUNT = 0
00000C 6E81   00009         MOVWF PORTB    ;send sum to PORTB SFR
              00010         END
```

在程序列表中,指令BNZ AGAIN的操作码是E1,相对地址是FDH。FDH也就是−3,所以地址偏移量是−3×2=−6。把相对地址−6加到下一条指令的地址0000CH上,可以得到目标地址−6+0CH=06H(舍去进位)。注意,000006是标签AGAIN的地址。FDH是负数,暗指程序将向后跳转。第5章将进一步讨论负数的加法运算。

尽管使用BNZ和DECF可以实现循环操作,可是,更好的方法是使用一条指令(如DCFSNZ),因为它把自减1和跳转简化在一条指令里。

107

3.1.7 无条件分支指令

无条件分支就是程序无条件跳转到目标地址。PIC18有两个无条件分支指令:GOTO(跳转到)和BRA(分支)。具体选用哪一条指令由目标地址决定。下面将分别介绍这两条指令。

1. GOTO指令(GOTO是长跳转指令)

GOTO是无条件转移指令,可以在PIC18的2 MB地址范围内任意转移。它是一个4B(32位)的指令,其中12位是操作码,剩下的20位表示20位的目标地址。20位的目标地址允许跳转的存储空间是1 MB(00000~FFFFFH),而不是2 MB。将程序计数器的最低位A0置为0,GOTO的20位目标地址就变成了地址位A21~A1。采用这样的办法,GOTO就可以覆盖整个00000~1FFFFH的2 MB地址范围,并且可以确保目标地址是偶地址。由于所有的PIC18指令都是2 B或者4 B的,GOTO不会转到某条指令的中间位置。请参阅图3-4。

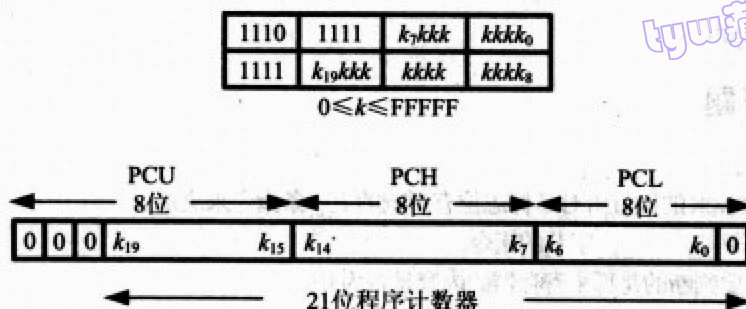


图 3-4 GOTO 指令

需要记住的是，尽管 PIC18 的程序计数器是 21 位的(ROM 地址范围是 2 MB)，可是并非所有的 PIC18 芯片都带有那么大的片上 ROM。有些 PIC18 芯片只有 4 KB~32 KB的片上程序 ROM，所以每个字节空间都是宝贵的。另外，PIC 还有一条 BRA(分支)指令，与 4 B 的 GOTO 指令相比，它只有 2 B。在 ROM 空间较小的许多应用设计中，这样能节省一些字节的存储空间。接下来介绍 BRA 指令。

2. BRA 指令(分支指令)

在这个 2B(16 位)的指令里，前 5 位是操作码，其余的 11 位是目标位置的相对地址。000~FFFH 的相对地址范围，又分为向前跳转和向后跳转；即相对于当前 PC 地址的－1024～＋1023的存储空间。如果是向前跳转，那么相对地址为正数；如果是向后跳转，那么相对地址就是负数。在这一点上，BRA 指令和条件分支指令很相似，只不过 BRA 使用的地址偏移量是 11 位而非 8 位。更多的技术细节如图 3-5 所示。

108



图 3-5 BRA(无条件分支)指令的地址范围

注意，BRA 是一个 2 B 的指令，相对于 GOTO 指令，它通常被优先考虑，因为它占用的 ROM 空间更小。有符号数的情况还会在第 5 章中讨论。

3.1.8 带有 $ 符号的 GOTO 指令

在没有监视程序的场合，人们常常使用 GOTO(跳转)指令转移到自身，以保持对微控制器的占用。一个简单的方法就是使用 $ 符号。对于如下的指令：

```
HERE  GOTO  HERE
```

可以用带有 $ 符号的指令来代替，即：

```
      GOTO  $
```

这种方法对 BRA 指令也同样适用。对于如下的指令：

```
OVER  BRA  OVER
```

依然可以用带有 $ 符号的指令来代替，即：

```
      BRA  $
```

这里面的 $ 表示同一行。

3.1.9　复习题

1. 助记符 BNZ 代表__________。
2. 判断对错：BNZ BACK 的执行由对 Z 标志位有影响的上一条指令来决定。
3. BNZ HERE 是一条__________字节的指令。
4. 在 JZ NEXT 中，要判断的是哪个寄存器的内容是否为 0?
5. GOTO 是一条__________字节的指令。

109

3.2　CALL（调用）指令和栈

另一个控制转移指令是用来调用子例程的 CALL 指令。子例程通常是一些需要频繁执行的任务所组成的程序段。除了节省存储空间外，子例程的使用可以让程序更加结构化。PIC18 有两个调用指令：CALL（长调用）和 RCALL（相对调用）。具体选用哪一个调用指令取决于目标地址。下面将分别介绍这两条指令。

3.2.1　CALL 指令

这是一条 4B(32 位)的指令，其中的 12 位用作操作码，其余的 20 位(A21～A1)用作子例程的地址。正如 GOTO 指令一样，程序计数器的最低位为 0，将自动地定位在偶地址上。因此，CALL 指令可以用来调用 PIC18 中 00000～1FFFFH 范围(2 MB)内任何地址的子例程，如图 3-6 所示。



图 3-6　CALL 指令

为了确保 PIC 在执行完调用的子例程后能准确返回，微控制器会自动把 CALL 下面的指令地址保存到栈里。当子例程被调用时，控制将会转移到该子例程，并且处理器把下一条指令的 PC 值保存到栈，然后开始从新的地址读取指令。在执行完子例程后，指令 RETURN 将把控制权还给调用者。每个子例程的最后一条指令都以 RETURN 结束。

3.2.2　PIC18 的栈和栈指针

栈是 CPU 用来临时存放一些非常重要的信息的读/写存储单元(RAM)。这些信息通常是地址，但也可以是数据。由于寄存器的数量是有限的，所以 CPU 很需要这些存储单元。PIC18 的程序计数器是 21 位的，因此它的栈也是 21 位的。也就是说，栈是使用 CALL 指令来

110

告诉PIC在调用子例程后应该返回到的位置的。同程序计数器一样,一个21位栈的地址空间是00000~1FFFFFH。如果栈是RAM,那么在CPU里需要有一个寄存器来指向它。用于访问栈的寄存器被称为SP(stack pointer,栈指针)寄存器。PIC18的栈指针是5位的,其取值范围是00~1FH。因此,它能提供32个地址,每个地址是21位宽,如图3-7所示。当PIC18通电时,SP寄存器的初始值为0。也就是说,栈的地址1是用于栈的第一个地址,因为SP指向上一次使用过的地址。这意味着栈的地址0是不可用的,这样,PIC18实际上只有31个栈地址。

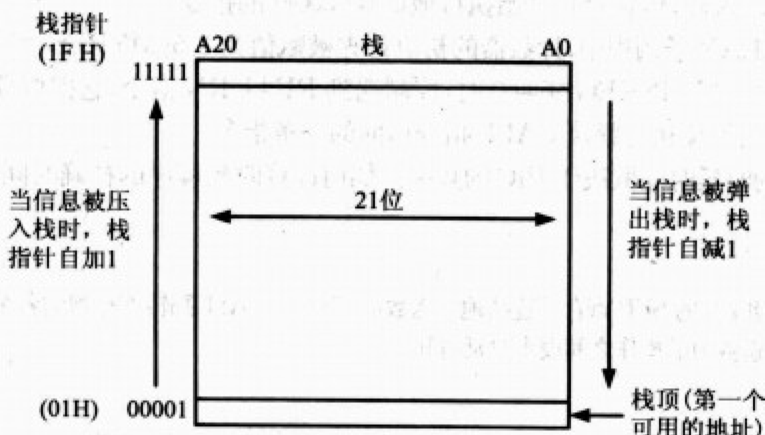


图3-7 PIC栈31×21

3.2.3 如何访问PIC18的栈

把CPU的信息(如程序计数器值)放入栈,被称为PUSH(压栈),而把栈信息取出并传送至CPU寄存器,则被称为POP(出栈)。换句话说,寄存器需要压栈来保存内容,而需要出栈来恢复内容。下面将描述栈操作的具体过程。

3.2.4 压栈

在PIC里,栈指针指向的是栈上一次用到的栈地址。上一次用到的栈地址也就是栈顶(TOS,top of the stack)。当数据被压入栈时,栈指针会自动加1。注意,这和很多其他的微处理器有所不同,如x86,当数据被压入栈时,它的SP是自动减1的。请看例3-9,每当CALL指令执行时,程序计数器的内容被压入栈,SP自动增加。注意,对于程序计数器的每次压栈操作,SP只增加1次。

3.2.5 出栈

出栈是把栈的内容送回到指定的寄存器(如程序计数器)的过程,这同压栈操作正好相反。当位于每个子例程最后的RETURN指令执行时,栈顶的值被复制到程序计数器,栈指针自动减小1次。因此,栈是LIFO(后进先出)的存储器。

3.2.6 CALL 指令和栈的作用

在 PIC 里,CPU 用栈来保存紧接 CALL 指令的下一条指令的地址。这就是 CPU 在执行完调用的子例程后能正确返回的原因。为了理解栈在微控制器中的重要性,下面将讨论例 3-9 的栈内容和栈指针。例 3-10 也有进一步的阐释。

对于例 3-9 中的程序,需要注意以下几点。

(1) 注意 DELAY 子例程。当执行第一条 CALL DELAY 时,它下面的一条指令 MOVLW 0xAA 的地址被压入栈,然后 PIC 开始执行地址 000300H 的指令。

(2) 在 DELAY 子例程中,计数器的初值首先被赋值为 255(MYREG=FFH),所以循环将被执行 256 次。当 MYREG 变为 0 时,控制遇到 RETURN 指令,它把栈顶的地址弹出到程序计数器,然后继续执行紧接 CALL 指令后面的一条指令。

例 3-9 中的时延时间取决于 PIC 的频率。如何计算时延程序的精确时间,将在本章的最后一节讨论。

例 3-9 向端口 *B* 的 SFR 寄存器连续地传送数值 55H 和 AAH,翻转寄存器的各个二进制位。在每两次向端口 *B* 传送数值的操作之间设置时延时间。

解:

```
MYREG EQU    0x08                      ;use location 08 as counter
      ORG      0
BACK  MOVLW    0x55                    ;load WREG with 55H
      MOVWF    PORTB                   ;send 55H to port B
      CALL     DELAY                   ;time delay
      MOVLW    0xAA                    ;load WREG with AA (in hex)
      MOVWF    PORTB                   ;send AAH to port B
      CALL     DELAY
      GOTO     BACK                    ;keep doing this indefinitely
;———— this is the delay subroutine
      ORG      300H                    ;put time delay at address 300H
DELAY MOVLW    0xFF                    ;WREG = 255,the counter
      MOVWF    MYREG
AGAIN NOP                              ;no operation wastes clock cycles
      NOP
      DECF     MYREG, F
      BNZ      AGAIN                   ;repeat until MYREG becomes 0
      RETURN                           ;return to caller
      END                              ;end of asm file
```


例 3-10 分析下面程序中的 CALL 指令的栈。

解: 当第一条 CALL 指令执行时,指令 MOVLW 0xAA 的地址被保存到栈(压栈)。被调用的子例程的最后一条指令必须是 RETURN,它告诉 CPU 把栈顶的地址弹出,并传送至 PC 寄存器,继续执行地址 000007 的指令。下图分别给出了 CALL 指令和 RETURN 指令执行后栈的结构。

```
LOC   OBJECT CODE      LINE SOURCE TEXT
  VALUE

                       00001 #DEFINE PORTB 0xF81
  00000008             00002 MYREG EQU 0x08       ;use location 08 as counter
```

```
                     00003
                     00004
  000000             00005         ORG     0
000000 0E55          00006  BACK   MOVLW   0x55    ;load WREG with 55H
000002 6E81          00007         MOVWF   PORTB   ;send 55H to port B
000004 EC80 F001     00008         CALL    DELAY   ;time delay
000008 0EAA          00009         MOVLW   0xAA    ;load WREG with AA (in hex)
00000A 6E81          00010         MOVWF   PORTB   ;send AAH to port B
00000C EC80 F001     00011         CALL    DELAY
000010 EF00 F000     00012         GOTO    BACK    ;keep doing this indefinitely
                     00013
                     00014 ;------- this is the delay subroutine
                     00015
  000300             00016         ORG     300H    ;put delay at address 300H
000300 0EFF          00017  DELAY  MOVLW   0xFF    ;WREG = 255,the counter
000302 6E08          00018         MOVWF   MYREG
000304 0000          00019  AGAIN  NOP             ;no op wastes clock cycles
000306 0000          00020         NOP
000308 0608          00021         DECF    MYREG, F
00030A E1FC          00022         BNZ     AGAIN   ;repeat until MYREG becomes 0
00030C 0012          00023         RETURN          ;return to caller
                     00024         END             ;end of asm file
```



3.2.7 栈的上限

如前所述,PIC18 只有 31 个栈空间,其地址范围是 01～1FH,栈地址 00 是不可用的。这就限制了一个程序里只能有 31 个调用。在 PIC 里,栈用于调用和中断。要记住的是,在调用子例程时,栈保存了 CPU 在执行完子例程后应该返回的位置。因此,要特别注意不要改变栈的内容。这在第 6 章将有更多的介绍。

3.2.8 在主程序里调用多个子例程

在汇编语言程序里,很多时候都会有一个主程序和很多供主程序调用的子例程,如图 3-8 所示。这样,每个子例程对应着一个独立的程序模块。每个程序模块可分别测试,然后与主程序结合在一起。更重要的是,在大型程序中,程序模块可以由不同的程序员来编写,这样可以缩短开发的时间。关于程序模块的介绍,请参阅第 6 章。

这里要强调的是,在使用 CALL 时,子例程的目标地址可以是 PIC18 的 2 MB 存储空间内的任何位置,程序实例请参阅例 3-11。但是,这种情况并不适用于另一个调用指令 RCALL,下面将解释其中的原因。

```
;MAIN program calling subroutines
            ORG     0
MAIN        CALL    SUBR_1
            CALL    SUBR_2
            CALL    SUBR_3

HERE        BRA     HERE         ;stay here
;---------end of MAIN
;
SUBR_1      ....
            ....
            RETURN
;---------end of subroutine 1
;
SUBR_2      ....
            ....
            RETURN
;---------end of subroutine 2

SUBR_3      ....
            ....
            RETURN
;---------end of subroutine 3
            END                  ;end of the asm file
```

图 3-8 调用子例程的 PIC 汇编主程序

例 3-11 编写程序,实现从 00 到 FFH 的加法,并把结果存放到 SFR 的端口 *B*。使用 CALL 指令,一个子例程用于把数据传送到端口 *B*,另外一个子例程用于时延。在每两次向端口 *B* 传送数据之间插入时延。

解:

```
LOC    OBJECT CODE  LINE   SOURCE TEXT
  VALUE
                    00001        list P=PIC18F458
                    00002  #include P18F458.INC
                    00003
  00000007          00004  COUNT  EQU    0x07   ;use location 07 for count-up
  00000008          00005  MYREG  EQU    0x08   ;use location 08 for delay
                    00006
000000              00007  ORG    0
000000 0E00         00008         MOVLW  0                ;WREG = 0
000002 6E07         00009         MOVWF  COUNT            ;count = 0
000004 EC06  F000   00010  BACK   CALL   DISPLAY
000008 EF02  F000   00011         GOTO   BACK
                    00012
                    00013  ;------ increment and put it in PORTB
00000C 2A07         00014  DISPLAY INCF  COUNT,F       ;increment count
00000E C007  FF81   00015         MOVFF  COUNT,PORTB   ;send it to PORTB
000012 EC80  F001   00016         CALL   DELAY
000016 0012         00017         RETURN               ;return to caller
                    00018
                    00019  ;------ this is the delay subroutine
```

```
000300             00020  ORG    300H   ;put time delay at address 300H
000300 0EFF        00021  DELAY  MOVLW  0xFF          ;WREG = 255, the counter
000302 6E08        00022         MOVWF  MYREG
000304 0000        00023  AGAIN  NOP    ;no operation wastes clock cycles
000306 0000        00024         NOP
000308 0000        00025         NOP
00030A 0608        00026         DECF   MYREG,F
00030C E1FB        00027         BNZ    AGAIN ;repeat until MYREG becomes 0
00030E 0012        00028         RETURN               ;return to caller
                   00029         END                  ;end of asm file
```



3.2.9 RCALL 指令(相对调用指令)

RCALL是一个2B的指令,这与4B的CALL指令不同。RCALL是2B指令,因此,子例程的目标地址必须在2KB的范围内,因为这2B指令中只有11位是用作地址的。在保存程序计数器到栈和执行RETURN指令方面,RCALL和CALL指令是没有区别的。唯一的区别是,CALL的目标地址可以是PIC18的2MB地址空间里任何一个,而RCALL的目标地址就只有2KB范围。在Microchip公司制造的PIC18芯片中,有些芯片的片上ROM只有4KB。在这种情况下,使用RCALL指令取代CALL指令可以节省大量的程序ROM空间。

114~115

当然,除了使用紧凑指令外,程序员还需要深入掌握给定微处理器指令,并能灵活地运用。请看下面的例3-12。

例3-12 请尽可能简练地重新编写例3-9的主要程序。

解:

```
        MYREG EQU 0x08
        ORG    0
        MOVLW 0x55              ;load WREG with 55H
BACK    MOVWF PORTB                     ;issue value in PORTB SFR
        RCALL DELAY             ;time delay
        COMPF PORTB,F                   ;complement Port B SFR
        BRA    BACK             ;keep doing this indefinitely
;————————this is the delay subroutine
DELAY   MOVLW 0xFF              ;WREG = 255, the counter
        MOVWF MYREG
AGAIN   NOP                     ;no operation wastes clock cycles
        NOP
        DECF   MYREG,F
        BNZ    AGAIN            ;repeat until MYREG becomes 0
        RETURN                  ;return to caller (MYREG = 0)
        END                     ;end of asm file
```

例 3-13 某开发商正使用 PIC18 微控制器芯片开发产品。该芯片只有 4 KB 的片上闪存 ROM。在对该芯片编程时，到底使用哪个指令更合适呢？CALL 还是 RCALL？

解：

RCALL 指令更合适，因为它是一个 2 B 的指令。在每次使用调用指令时，它可以节省 2 B 的空间。但是，如果目标地址范围超过了 2 KB，那么就只能使用 CALL 指令。

3.2.10 复习题

1. PIC18 栈的数据宽度是多少？
2. 判断对错：对于 PIC18，可以使用 CALL 指令在 2 MB 的代码空间里任意地转移控制。
3. CALL 指令是一个_____字节的指令。
4. 判断对错：对于 PIC18，可以使用 RCALL 指令在 2 MB 的代码空间里任意地转移控制。
5. 每次使用 CALL 指令时，栈指针 SP 是_____(自动递增，自动递减)的。
6. 每次使用 RETURN 指令，栈指针 SP 是_____(自动递增，自动递减)的。
7. 在通电后，PIC 使用地址_____作为栈顶。
8. PIC18 提供的栈数量是多少？
9. RCALL 指令是一个_____字节的指令。
10. 指令_____(RCALL，CALL)占用的 ROM 空间更多。

116

3.3 PIC18 的时延与指令流水线

上一节已经涉及了 DELAY 子例程。本节将讨论 PIC18 如何产生各种不同的时延以及时延的精确计算。此外，还会讨论指令流水线及其对执行时间的影响。

3.3.1 PIC18 的时延计算

在使用汇编语言创建时延程序时，必须注意影响时延精确性的两个因素。

(1) 晶振频率。输入引脚 OSC1 和 OSC2 的晶振频率是影响时延计算的一个因素。对于指令周期，时钟周期的持续时间就是晶振频率的作用。

(2) PIC 的设计。从 20 世纪 70 年代开始，IC 工艺和微处理器的结构化设计都有了巨大的发展。由于多年来 IC 工艺的局限和 CPU 设计经验的不足，指令周期时间都比较长。在 20 世纪 80 年代和 90 年代，IC 工艺和 CPU 设计都取得了进展，所研制出的单指令周期后来成为许多微控制器的共有属性。事实上，要提高微控制器的性能，而又不失同旧系列的微控制器的代码兼容性，一个有效的方法就是减少指令执行所需的指令周期数。人们不禁会问，微控制器(如 PIC)怎么才能在一个指令周期里执行完一条指令呢？这里有 3 个解决办法：(1)使用哈佛结构，将最大数量的代码和数据传送至 CPU；(2)利用 RISC 结构特性，如固定的指令长度；(3)使用流水线技术，指令的读取和执行重叠进行。哈佛结构和 RISC 结构已在第 2 章介绍过，下面将介绍流水线技术。

3.3.2 流水线

早期的微处理器(如 8085)中，CPU 在给定时间里要么读取指令，要么执行指令。换而言

之,CPU首先要从存储空间里读取指令,然后执行它,再读取下一条指令,再执行,依此类推。最简单的流水线的思想就是让CPU的读取和执行能同时进行,如图3-9所示。



图 3-9 流水线与非流水线的对比

3.3.3 PIC的指令周期时间

众所周知,CPU需要花费一定的时间来执行指令。在PIC中,这个时间被称为指令周期(有些CPU称为机器周期)。由于PIC18的所有指令都是2B和4B的,所以大多数指令的执行都不超过一个或者两个指令周期。(注意,有些指令的执行可能需要更多的指令周期,如执行BTFSS需要3个指令周期。)附录A罗列了PIC18的指令清单和它们的指令周期。对于PIC系列,指令周期时间的长短取决于PIC系统的晶振频率。晶体振荡器和片上时钟电路为PIC的CPU提供时钟信号源(请参阅第8章)。在PIC18里,一个指令周期由4个晶振周期组成。因此,要计算PIC的指令周期,就得先计算出1/4的晶振频率,再取倒数,如例3-14所示。

例 3-14 根据下面给出的3个基于PIC的系统的晶振频率,计算每一个系统的指令周期。

(a) 4 MHz (b) 16 MHz (c) 20 MHz

解:

(a) 4/4=1 MHz,指令周期是 1/1 MHz =1 μs

(b) 16 MHz/4=4 MHz,指令周期是 1/4 MHz=0.25 μs=250 ns

(c) 20 MHz/4=5 MHz,指令周期是 1/5 MHz=0.2 μs=200 ns

3.3.4 分支代价

今天,指令的重叠读取与执行在微控制器(如PIC)里已得到广泛的使用。在采用流水线作业时,需要使用缓冲器或者队列来存放预读取的指令和执行准备就绪的指令。在一些情况下,CPU必须要冲刷队列。举个例子,当分支指令执行时,CPU从新存储地址读取代码,而队列中之前读取的代码就会被丢弃。在这种情况下,执行单元必须等待,直到取值单元读取到新的指令。这就叫作分支代价(branch penalty)。这种代价是额外的指令周期,它用来读取目标地址的指令,而不是直接执行分支指令后面的指令。要记住的是,当CPU要转移到其他地址的时候,分支指令下面的指令已经被读取并且在队列的下一行等待执行。这意味着,当大多数的PIC指令只需要1个指令周期时,而有一些指令则会占用2~3个指令周期。它们是GOTO、BRA、CALL和所有的条件分支指令(如BNZ、BC等)。当条件分支没有实现跳转的

时候，它只占用1个指令周期。例如，BNZ指令在Z=0的时候将会执行跳转，它占用2个指令

118 周期。当Z=1时，跳转不起作用，它只占用1个指令周期。请看例3-15和例3-16。

例 3-15 对于一个4 MHz的PIC18系统，计算下面指令的执行时间。

```
(a) MOVLW        (b) DECF        (c) MOVWF
(d) ADDLW        (e) NOP         (f) GOTO
(g) CALL         (h) BNZ
```

解：4 MHz系统的机器周期为1 μs，如例3-14所示。附录A又列出了上面每条指令的指令周期。因此，可以得到：

指令	指令周期	执行时间
(a) MOVLW 0x55	1	1 ×1 μs = 1 μs
(b) DECF MYREG	1	1 ×1 μs = 1 μs
(c) MOVWF	1	1 ×1 μs = 1 μs
(d) ADDLW	1	1 ×1 μs = 1 μs
(e) NOP	1	1 ×1 μs = 1 μs
(f) GOTO	2	2 ×1 μs = 2 μs
(g) CALL	2	2 ×1 μs = 2 μs
(h) BNZ	2/1	(2 μs，失败的话只需1μs)

例 3-16 确定下面代码段的延迟时间，假设晶振频率为4 MHz。

解：由附录A，可知下面时延子例程中各条指令的机器周期。

```
                                                指令周期
MYREG EQU    0x08          ;use location 08 as counter

DELAY    MOVLW     0xFF                            1
         MOVWF     MYREG                           1
AGAIN    NOP                                       1
         NOP                                       1
         DECF      MYREG,F                         1
         BNZ       AGAIN                           2

         RETURN                                    1
```

因此，可得到总的时延时间为 [(255×5)+1+1+1]×1 μs=1278 μs。

注意，若BNZ往回跳，则占用2个指令周期；若跳出循环，则只占1个周期。也就是说，上面结果应该

119 为1277 μs。

3.3.5 PIC18 的时延计算

从上一节可以看到，一个时延子例程由两部分构成：设置计数器初值，循环。大多数的时延都是由循环体实现的，如例3-17和例3-18所示。

例 3-17 确定下面程序的时延时间，假设晶振频率为4 MHz。

```
MYREG EQU    0x08               ;use location 08 as counter

         ORG      0
BACK     MOVLW    0x55          ;load WREG with 55H
         MOVWF    PORTB         ;send 55H to port B
         CALL     DELAY         ;time delay
```

```
        MOVLW    0xAA        ;load WREG with AA (in hex)
        MOVWF    PORTB       ;send AAH to port B
        CALL     DELAY
        GOTO     BACK        ;keep doing this indefinitely

;------ this is the delay subroutine
        ORG      300H        ;put time delay at address 300H
DELAY   MOVLW    0xFA        ;WREG = 250, the counter
        MOVWF    MYREG
AGAIN   NOP                  ;no operation wastes clock cycles
        NOP
        NOP
        DECF     MYREG, F
        BNZ      AGAIN       ;repeat until MYREG becomes 0
        RETURN               ;return to caller
        END                  ;end of asm file
```

解:从附录 A 可知下面 DELAY 子例程的各指令的机器时钟。

```
                                 指令周期
DELAY   MOVLW    0xFA              1
        MOVWF    MYREG             1
AGAIN   NOP                        1
        NOP                        1
        NOP                        1
        DECF     MYREG, F          1
        BNZ      AGAIN             2

        RETURN                     1
```

因此,得到的总时延时间是$[(250\times6)+1+1+1]\times1\ \mu s=1503\ \mu s$。

通常,根据循环体内的指令来计算时延时间,而忽略循环体外的指令的时钟周期。 120

在例 3-16 里,MYREG 寄存器的最大值是 225。因此,一个增加时延的方法是在循环中使用 NOP。NOP(NO Operation,空操作)仅仅是消耗时钟,但是它却占用 2 B 的程序 ROM 空间,这对于完成一个指令周期的代价实在太大。更好的办法是使用嵌套循环。

3.3.6 时延的嵌套循环

另一个实现较大时延的方法是在一个循环中再放入一个循环,也就是所谓的*循环嵌套*,如例 3-18 所示。对比例 3-19,将会发现多次使用 NOP 指令的缺点。

例 3-18 假设指令周期为 1 μs,试确定下面子例程的时延时间。

```
R2      EQU     0x7
R3      EQU     0x8
DELAY                              指令周期
        MOVLW       D'200'           1
        MOVWF       R2               1
AGAIN   MOVLW       D'250'           1
        MOVWF       R3               1
HERE    NOP                          1
        NOP                          1
```

```
        DECF        R3, F       1
        BNZ         HERE        2
        DECF        R2, F       1
        BNZ         AGAIN       2
        RETURN                  1
```

解:循环 HERE 的时延时间为(5×250)1 μs=1250 μs。循环 AGAIN 重复执行了循环 HERE 200 次;因此,总的时延时间为 200×1250 μs=250 000 μs,这里没有包括前面的程序指令。然而,还需要增加外层循环的下列指令的执行时间:

```
AGAIN       MOVLW       D'250'      1
            MOVWF       R3          1
            .....
            DECF        R2, F       1
            BNZ         AGAIN       2
```

在循环 AGAIN 前后的上述指令将增加 5×200×1 μs=1000 μs 的时延。同时,在指令 BNZ HERE 没有执行转移的时候,还应该减去 200 μs。因此,在上面的 DELAY 子例程里,所得到的时延时间是 250 000+1000−200=250 800 μs=250.8 ms。注意,嵌套循环的时延时间是一个近似值,这同其他时延循环一样,因为人们一般会忽略子例程开始处的几条指令和子例程的结束指令 RETURN。NOP 是一个 2 B 的指令。在上面的 DELAY 子例程里一共使用了 11 条指令,而这些指令都是 2 B 的。也就是说,循环时延程序占用了 22 B 的 ROM 代码空间。

121

例 3-19　假设指令周期为 1 μs,试确定下面子例程的时延时间。对比例 3-18,指出其缺点。

```
MYREG EQU    0x8

                                机器时钟
DELAY     MOVLW     D'200'          1
          MOVWF     MYREG           1

AGAIN     NOP                       1
          NOP                       1
          NOP                       1
          NOP                       1
          NOP                       1
          NOP                       1
          NOP                       1
          NOP                       1
          NOP                       1
          NOP                       1
          NOP                       1
          NOP                       1
          DECF      MYREG, F        1
          BNZ       AGAIN           2

          RETURN                    1
```

解:循环 AGAIN 内的时延时间是[200(13+2)]×1 μs=3000 μs。NOP 是一个 2 B 的指令,但是它只起到消耗时钟的作用。在上面的 DELAY 程序里共有 17 条指令,所有的指令都是 2 B 的。也就是说,循环时延占用了 34 B 的 ROM 代码空间,却只能产生 3000 μs 的时延。这就是为什么使用嵌套循环来代替 NOP 指令产生时延的原因所在。第 9 章将讨论怎样使用 PIC 定时器来更有效地产生时延。

从上述的讨论中可以得出结论:用指令来产生时延并不是最可靠的方法。为了得到更准确的时延,这需要用到定时器,请参阅第9章。可以使用MPLAB仿真器来验证时延时间和指令周期数。同时,要想获得PIC微控制器中准确的时延,必须使用示波器来测量精确的时延。 122

例3-20 编写程序,每隔一秒翻转一次SFR的PORTB的各二进制位。假设晶振频率是10 MHz,系统使用的是PIC18F458。

解:

```
;tested using MPLAB with PIC18F458 operating at 10 MHz
R2      EQU     0x2
R3      EQU     0x3
R4      EQU     0x4

        MOVLW   0x55            ;load WREG with 55H
        MOVWF   PORTB           ;send 55H to PORTB B
BACK    CALL    DELAY_500MS     ;time delay
        COMF    PORTB           ;complement PORTB
        GOTO    BACK            ;keep doing this indefinitely

;-------  this is the delay subroutine
DELAY_500MSEC
            MOVLW       D'20'
            MOVWF       R4
BACK        MOVLW       D'100'
            MOVWF       R3
AGAIN       MOVLW       D'250'
            MOVWF       R2
HERE        NOP
            NOP
            DECF        R2, F
            BNZ         HERE
            DECF        R3, F
            BNZ         AGAIN
            DECF        R4, F
            BNZ         BACK
            RETURN
```

时延时间:$20\times100\times250\times5\times400$ ns=1 000 000 000 ns=1 000 000 μs=1 s

上述时间的计算,没有包括两个外层循环有关的引导程序。使用MPLAB仿真器可以验证时延。

3.3.7 PIC多级执行流水线

使用超级流水线技术可以加快指令的执行速度。在超级流水线里,指令的执行过程被分解成很多可以并行进行的小步骤。这样,很多指令的执行都是重叠的。超级流水线的一个缺点是,执行的速度受最慢任务的限制。拿做比萨饼来打个比方。你可以把做比萨饼的过程分成很多步骤,例如擀面、放料和烘烤,但是过程是受到最慢的烘烤步骤的限制的,无论其他步骤进行得如何快。那如果用2个或3个烤箱来烘烤比萨饼以加快制作速度呢?这对于做比萨饼是有效的,不过,不适用于执行程序,因为在指令的执行过程中需要确保指令的顺序是完整的,而且没有跳跃执行的情况。对于PIC18,执行单元占用4个时钟周期,如图3-10所示。 123



图 3-10　指令读取后的流水线活动

图 3-10 解释了为什么把晶振频率除以 4 来获取指令周期。在 Q1 段，对已经读取并等候在队列里的指令进行译码。在 Q2 段，从文件寄存器里读取操作数。在 Q3 段，执行操作：完成两数相加。在 Q4 段，结果写入目的寄存器。在实际中，可以构造适用于 4 个指令的 PIC18 超级流水线，如图 3-11 所示。

注意，在很多计算机结构的书籍中，处理过程就是执行，而写操作则称作回写。

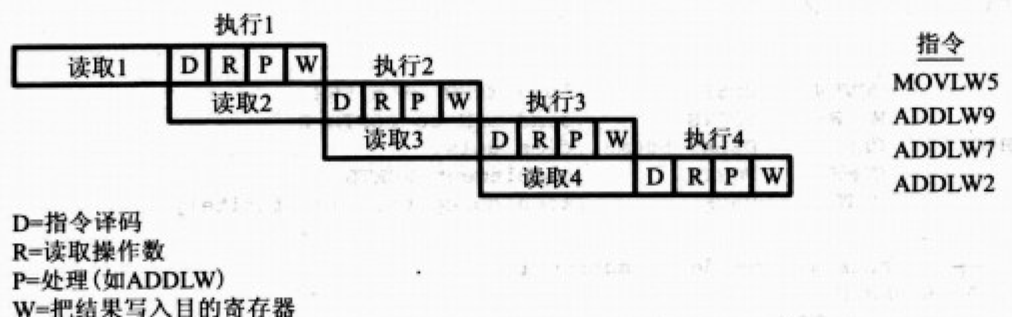


图 3-11　流水线的获取和执行操作

3.3.8　复习题

1. 判断对错：在 PIC18 里，指令周期持续 4 个时钟周期。
2. 执行 PIC18 指令需要的最小指令周期是______。
3. 在第 2 题中，需要的最大周期是什么？是哪条指令？
4. 确定晶振频率为 12 MHz 时的指令周期。
5. 假设晶振频率为 4 MHz。确定下面 DELAY 子例程的循环部分的时延。

```
DELAY
        MOVLW    D'100'
        MOVWF    MYREG
HERE    NOP
        NOP
        NOP
        NOP
        NOP
        DECF     MYREG, F
        BNZ      HERE
        RETURN
```

6. 判断对错：在 PIC18 里，指令周期持续 6 个晶振频率周期。
7. 确定 PIC18 的机器周期，假设晶振频率为 8 MHz。
8. 判断对错：在 PIC 里指令的读取和执行是同时进行的。
9. 判断对错：BRA 和 CALL 指令都占用 2 个指令周期。
10. 判断对错：BNZ 指令总是占用 2 个指令周期。

小结

程序的执行是从一条指令到下一条指令按顺序进行的，除非遇到控制转移指令。汇编程序里的控制转移有很多，包括条件分支和无条件分支，还有调用指令。

在 PIC 汇编语言中，循环的执行需要使用指令来实现计数器的递减，在计数器不为 0 时，跳转到循环体的顶部。这个过程可以由 BNZ 指令来完成。其他的条件分支指令根据进位标志位、零标志位和状态寄存器的其他标志位实现转移控制。无条件分支指令可以是长跳转也可以是短跳转，这取决于目标地址的位置。特别要注意，CALL 指令和 RCALL 指令对于栈的影响。

习题

1. 在 PIC 里，指令 `BNZ target` 循环地执行的最大次数是______。
2. 如果条件分支没有执行，那么下一条被执行的指令是什么？
3. 在计算分支的目标地址时，需要将位移量加到寄存器______。
4. 伪指令 BRA 代表______，这是一个______字节的指令。
5. GOTO 指令是一个______字节的指令。
6. 用 BRA 取代 GOTO 的优势是什么？
7. 判断对错：BNZ 的目标地址可以是 2 MB 的地址空间。
8. 判断对错：所有 PIC 分支指令都可以跳转到 2 MB 的地址空间。
9. 下面哪个条指令是 2 B 指令？
   (a)BZ　　(b)BNC　　(c)GOTO　　(d)BRA
10. 剖析 BRA 指令，说明分别有多少位是用于操作数和操作码的，以及它可以转移的地址空间大小。
11. 判断对错：所有的条件分支都是 2 B 指令。
12. 编写代码，实现 1000 次的循环嵌套。
13. 编写代码，实现 100 000 次的循环嵌套。
14. 试确定下面程序执行的循环次数。

```
        MOVLW   D'200'
        MOVWF   REGA
BACK    MOVLW   D'100'
        MOVWF   REGB
HERE    DECF    REGB,F
        BNZ     HERE
        DECF    REGA,F
        BNZ     BACK
```

15. 若操作码的第二字节是______（负数，正数），则 BNZ 的目标地址是向后的。
16. 若操作码的第二字节是______（负数，正数），则 BNZ 的目标地址是向前的。
17. CALL 是一个______字节的指令。
18. RCALL 是一个______字节的指令。
19. 判断对错：RCALL 的目标地址可以是 2 MB 地址空间里的任何位置。

20. 判断对错:CALL 的目标地址可以是 2 MB 地址空间里的任何位置。
21. 当 CALL 指令执行时,会用到栈里多少个位置?
22. 当 RCALL 指令执行时,会用到栈里多少个位置?
23. 系统重启后,栈的第一个有效地址是______。
24. 试描述 RETURN 指令的有关操作。
25. 请指出 PIC18 的栈大小。
26. 对于 PIC18,当调用指令执行时,哪个地址会被压栈,栈指针又如何变化?
27. 当指令周期为 1.25 μs 时,请确定晶振频率。
28. 当晶振频率为 20 MHz 时,请确定指令周期。
29. 当晶振频率为 10 MHz 时,请确定指令周期。

30. 当晶振频率为 16 MHz 时,请确定指令周期。
31. 判断对错:在 CALL 和 RCALL 中,尽管一个是 4 B 指令,另一个是 2 B 指令,但是它们所占用的执行时间是相同的。
32. 假设 PIC18 系统的频率为 4 MHz,试计算下面的时延子例程的时延时间。

```
         MOVLW    D'200'
         MOVWF    REGA
BACK     MOVLW    D'100'
         MOVWF    REGB
HERE     NOP
         DECF     REGB,F
         BNZ      HERE
         DECF     REGA,F
         BNZ      BACK
```

33. 假设 PIC18 系统的频率为 16 MHz,试计算下面的时延子例程的时延时间。

```
         MOVLW    D'200'
         MOVWF    REGA
BACK     MOVLW    D'100'
         MOVWF    REGB
HERE     NOP
         NOP
         DECF     REGB,F
         BNZ      HERE
         DECF     REGA,F
         BNZ      BACK
```

34. 假设 PIC18 系统的频率为 4 MHz,试计算下面的时延子例程的时延时间。

```
         MOVLW D'200'
         MOVWF REGA
BACK     MOVLW D'250'
         MOVWF REGB
HERE     NOP
         DECF     REGB
         BNZ      HERE
         DECF     REGA
         BNZ      BACK
```

35. 假设 PIC18 系统的频率为 10 MHz，试计算下面时延子例程的时延时间。

```
        MOVLW   D'200'
        MOVWF   REGA
BACK    MOVLW   D'100'
        MOVWF   REGB
        NOP
        NOP
        NOP
HERE    DECF    REGB,F
        BNZ     HERE
        DECF    REGA,F
        BNZ     BACK
```

127

复习题答案

3.1 节

1. 若非零，则跳转　　2. 正确。　　3. 2
4. 状态寄存器的 Z 标志位。　　5. 4

3.2 节

1. 21 位。　　2. 正确。　　3. 4　　4. 错误。　　5. 自加 1
6. 自减 1　　7. 1　　8. 31 个地址(21×31)。　　9. 2　　10. CALL

3.3 节

1. 正确。　　2. 1　　3. 2 和 CALL。同样，DECFSZ 指令占用 3 个指令周期。
4. 12 MHz/4=3 MHz，MC=1/3 MHz=0.33 μs　　5. [100(1+1+1+1+1+1+2)]×1 μs=0.8 ms
6. 错误。它占用 4 个时钟周期。7. 8MHz/4=2MHz。机器周期为 1/2MHz=500ns。
8. 正确。　　9. 正确。　　10. 错误。仅当它跳转到目标地址。

128

第 4 章 PIC I/O 端口编程

学习目标：

- ☐ PIC18 的所有端口
- ☐ PIC18 引脚的双重功能
- ☐ 输入输出的汇编语言编程
- ☐ 端口 *A*、*B*、*C* 和 *D* 的双重功能
- ☐ 用作 I/O 处理的 PIC 指令编程
- ☐ PIC 端口的位寻址能力

129

本章将通过许多例子来说明 PIC18 的 I/O 端口编程技术。4.1 节将介绍如何使用字节数据访问 I/O 端口，4.2 节将详细讨论端口的位操作。

4.1 PIC18 的 I/O 端口编程

在 PIC18 系列中，用于 I/O 操作的端口有很多，这取决于你所选择的 PIC 芯片。图 4-1 描述了 40 个引脚的 PIC18F458 芯片。这里共有 33 个引脚供 5 个端口使用，它们分别是 PORTA、PORTB、PORTC、PORTD 和 PORTE。其余的引脚分别被用作V_{dd}(V_{cc})、V_{ss}(GND)、OSC1、OSC2、MCLR(复位)以及一组引脚 V_{dd}和 V_{ss}，这将在第 8 章中讨论。

4.1.1 I/O 端口引脚及其功能

PIC18 系列的端口数量取决于芯片上的引脚数。18 个引脚的 PIC18 芯片只有端口 *A* 和 *B*，而 64 个引脚的 PIC18 芯片则有端口 *A* 到端口 *J*，80 个引脚的 PIC18 芯片有端口 *A* 到端口 *L*，如表 4-1 所示。40 引脚的 PIC18F458 有 5 个端口，即 PORTA、PORTB、PORTC、PORTD 和 PORTE。要将这些端口作为输入或者输出，必须对它们进行初始化编程，这是本节将详细讨论的部分。除了用作简单的 I/O 端口外，每个端口还有其他的功能，如 ADC、定时器、中断、串行通信等引脚。图 4-1 同样也标出了 PIC18F458 引脚的第二功能。有关引脚的第二功能将在后面的章节讨论，本章主要讲述 PIC18 系列的简单的 I/O 功能。不是所有的端口都有 8 个引脚。比如，PIC18F458 的端口 *A* 有 7 个引脚，端口 *B*、*C*、*D* 都有 8 个引脚，而端口 *E* 却只有 3 个引脚。每个端口都有 3 个 SFR，如表 4-2 所示，它们分别被指定为 PORTx、TRISx 和 LATx 。比如，对于端口 *B*，3 个相关的 SFR 是 PORTB、TRISB 和 LATB。注意，TRIS 代表 TRIState(三态)，而 LAT 代表 LATch (锁存器)。接下来将介绍如何访问这些与端口有关的 SFR。

130

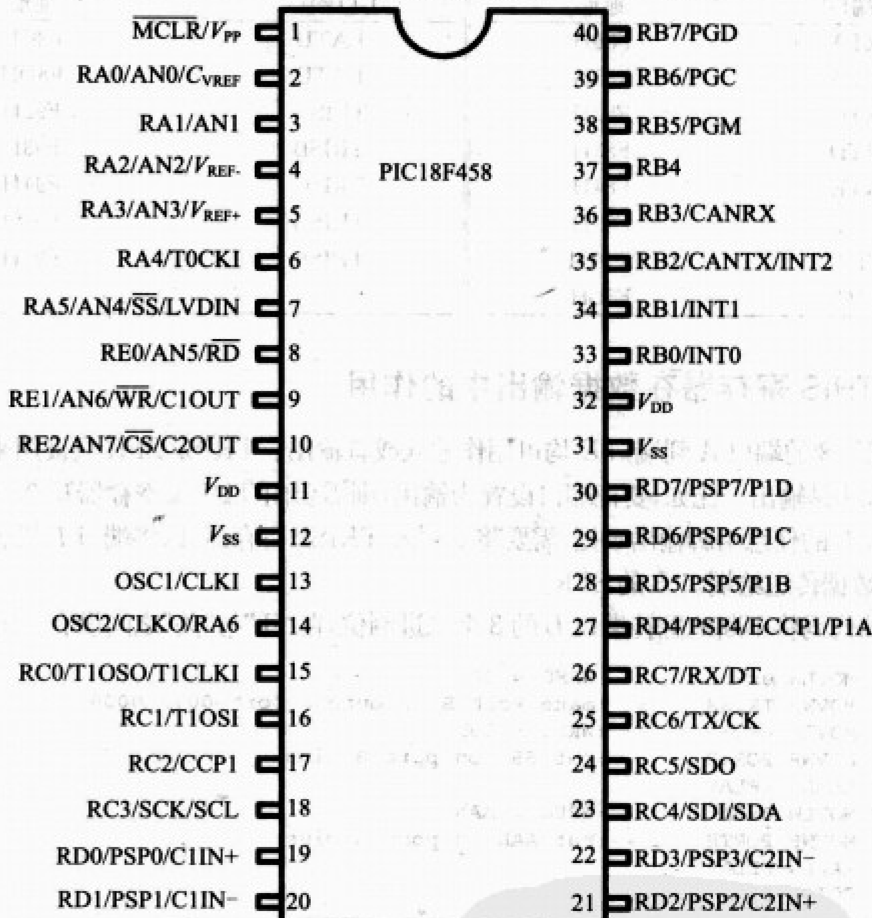


图 4-1 PICF458 引脚图

表 4-1 PIC18 系列单片机的 I/O 引脚数

引脚数	18 脚	28 脚	40 脚	64 脚	80 脚
芯片型号	PIC18F1220	PIC18F2220	PIC18F458	PIC18F6525	PIC18F8525
A 端口	X	X	X	X	X
B 端口	X	X	X	X	X
C 端口		X	X	X	X
D 端口			X	X	X
E 端口			X	X	X
F 端口				X	X
G 端口				X	X
H 端口				X	X
J 端口				X	X
K 端口					X
L 端口					X

注意：X 表示该 I/O 可用。

表 4-2 PIC18F458 I/O 的 SFR 地址

I/O 端口	地址	I/O 端口	地址
PORTA	F80H	LATD	F8CH
PORTB	F81H	LATE	F8DH
PORTC	F82H	TRISA	F92H
PORTD	F83H	TRISB	F93H
PORTE	F84H	TRISC	F94H
LATA	F89H	TRISD	F95H
LATB	F8AH	TRISE	F96H
LATC	F8BH		

4.1.2 TRIS 寄存器在数据输出中的作用

PIC18F458 的端口 *A* 到端口 *E* 均可用作输入或者输出。TRISx SFR 只被用来指定一个端口是输入还是输出。比如，要把端口设置为输出，则需要向 TRISx 寄存器写 0。换而言之，为了从端口 *B* 的任意引脚输出数据，需要将 0 写入 TRISB 寄存器，以将端口 *B* 设置为输出端口，然后将数据传送给端口 *B* 的 SFR。

131

以下的代码将不断地翻转端口 *B* 的 8 个二进制位，在“开”与“关”之间设有一定的时延。

```
        MOVLW 0x0       ;WREG = 00
        MOVWF TRISB     ;make Port B an output port 0000 0000
L1      MOVLW 0x55      ;WREG = 55h
        MOVWF PORTB     ;put 55h on port B pins
        CALL  DELAY
        MOVLW 0xAA      ;WREG = AAh
        MOVWF PORTB     ;put AAh on port B pins
        CALL  DELAY
        GOTO  L1
```

值得注意的是，除非激活 TRIS 位（将它置 0），否则数据将不能从端口寄存器输出到 PIC 的引脚。也就是说，如果将以上代码的前两行删除，55H 和 AAH 的值将不会在引脚输出。它们将被存储在 CPU 内的端口 *B* 的 SFR 中。

分析图 4-3 和图 4-4，可以清楚地知道 TRISx 寄存器在将数据从 PORTx 输出到引脚的过程中所起的作用。如果你对逻辑门的内部结构不太熟悉，请参阅附录 C。注意，在图 4-2 中，CMOS 的“开”和“关”状态均取自附录 C。

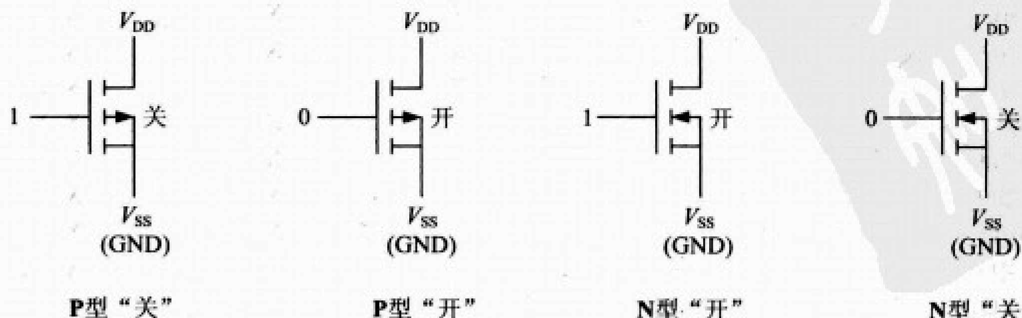


图 4-2 P 型和 N 型 CMOS 管的状态

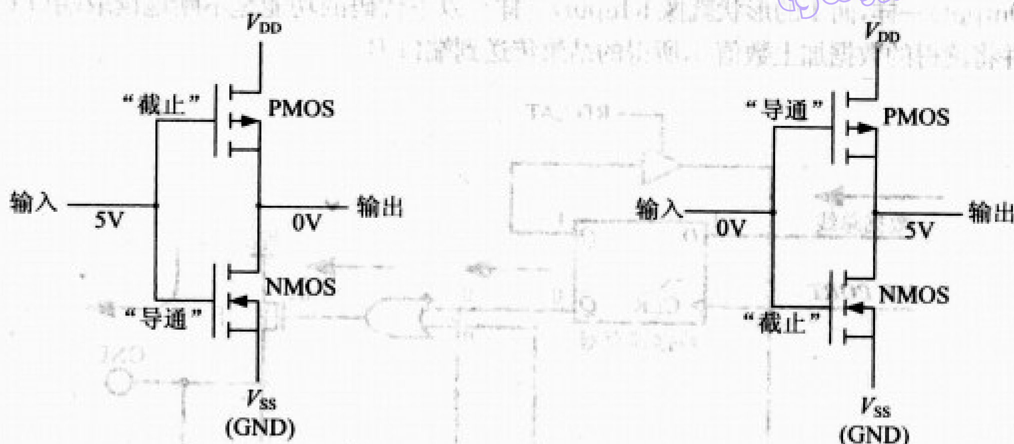


图 4-2 （续）

132

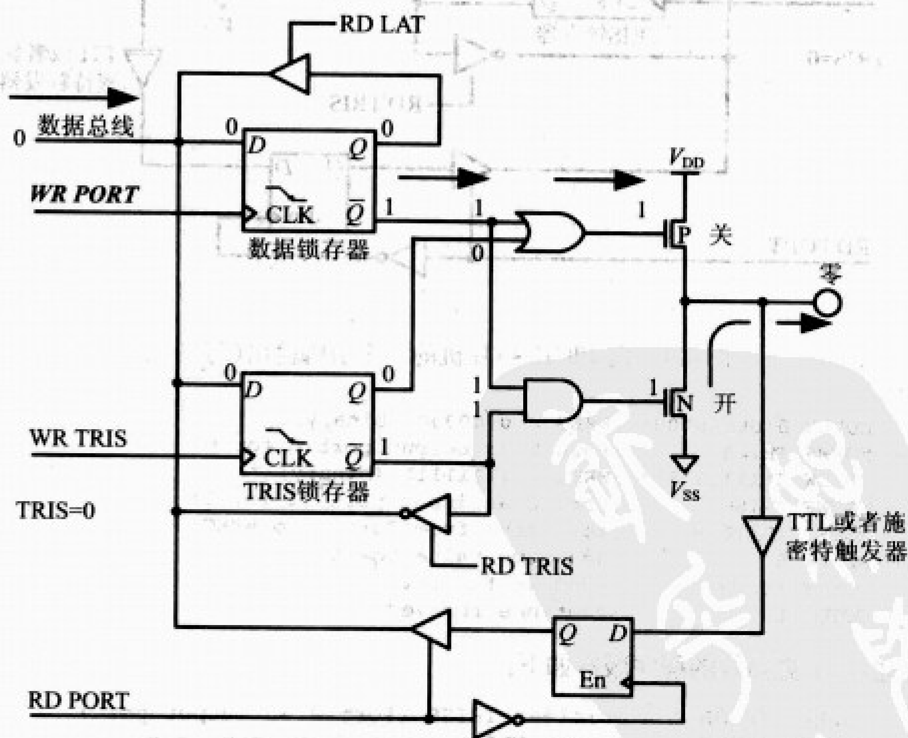


图 4-3 向 PIC18 单片机的一个引脚输出（写）0

注意，当系统复位时，所有端口的 TRISx 寄存器的值都是 FFH。这正如后面将要看到的，所有的端口都被设置成了输入端口。

133

4.1.3 TRIS 寄存器在数据输入中的作用

为了把端口设置为输入端口，必须先向该端口的 TRISx 寄存器写 1，然后才能读取引脚上的数据。注意，这里 0 代表输出，1 代表输入。这是很容易记住的，因为 0 的形状就像

O(Output)一样,而1的形状就像I(Input)一样。以下代码的功能是不断地读取端口C的引脚,并将读得的数据加上数值5,所得的结果传送到端口B。

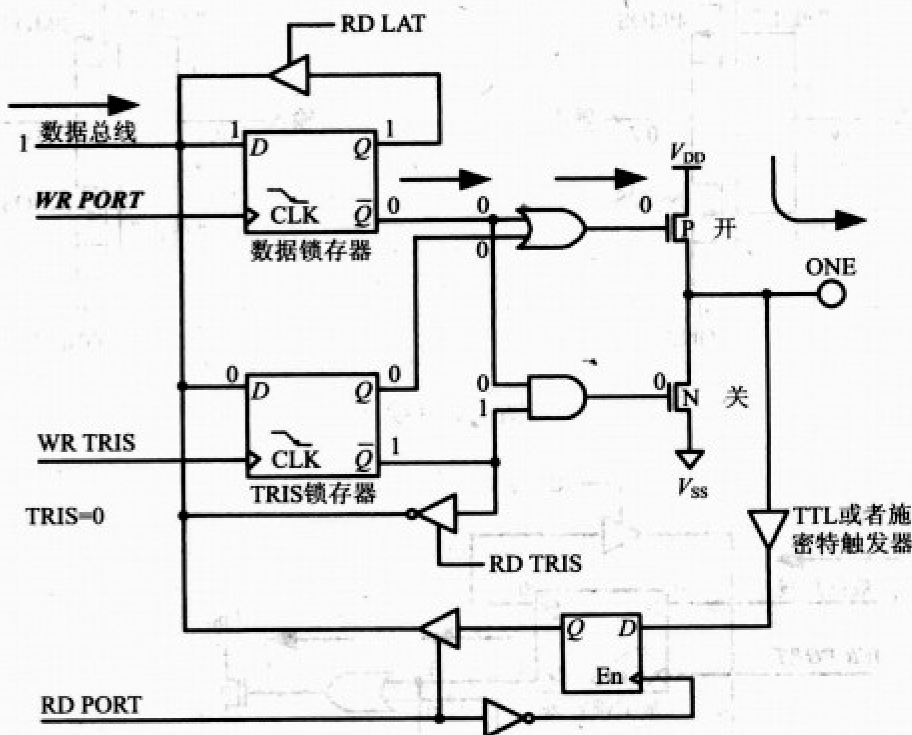


图4-4 向PIC18单片机的一个引脚输出(写)1

```
        MOVLW B'00000000' ;WREG = 0000000 (binary)
        MOVWF TRISB       ;Port B an output port(0 for O)
        MOVLW B'11111111' ;WREG = 1111111 (binary)
        MOVWF TRISC       ;Port C an input port (1 for I)
L2      MOVF  PORTC,W     ;move data from Port C to WREG
        ADDLW 5           ;add some value to it
        MOVWF PORTB       ;send it to Port B
        GOTO  L2          ;continue forever
```

对应地,一个更高效的程序版本如下:

```
        CLRF  TRISB       ;clear TRISB (Port B an output port)
        SETF  TRISC       ;set TRISC (Port C an input port)
L2      MOVF  PORTC,W     ;get data from port C
        ADDLW 5           ;add some value
        MOVWF PORTB       ;send it to port B
        BRA   L2
```

需要再次注意的是,除非激活TRIS位(将它置1),否则数据将不能从端口C的引脚传送到WREG寄存器。分析图4-5和图4-6,可以知道TRISX寄存器在允许数据从引脚读到CPU中的作用。

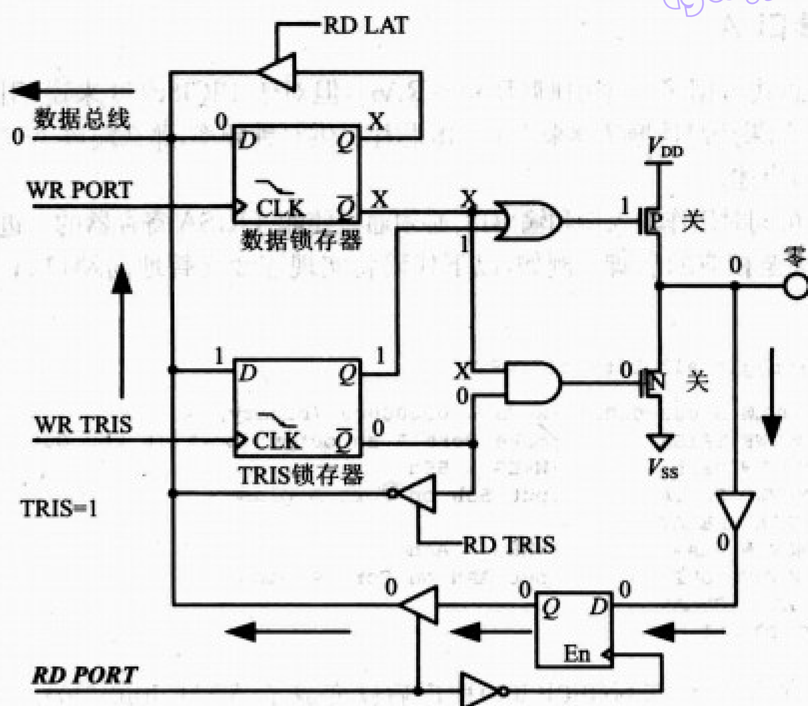


图 4-5 从 PIC18 单片机的一个引脚输入(读)0

134

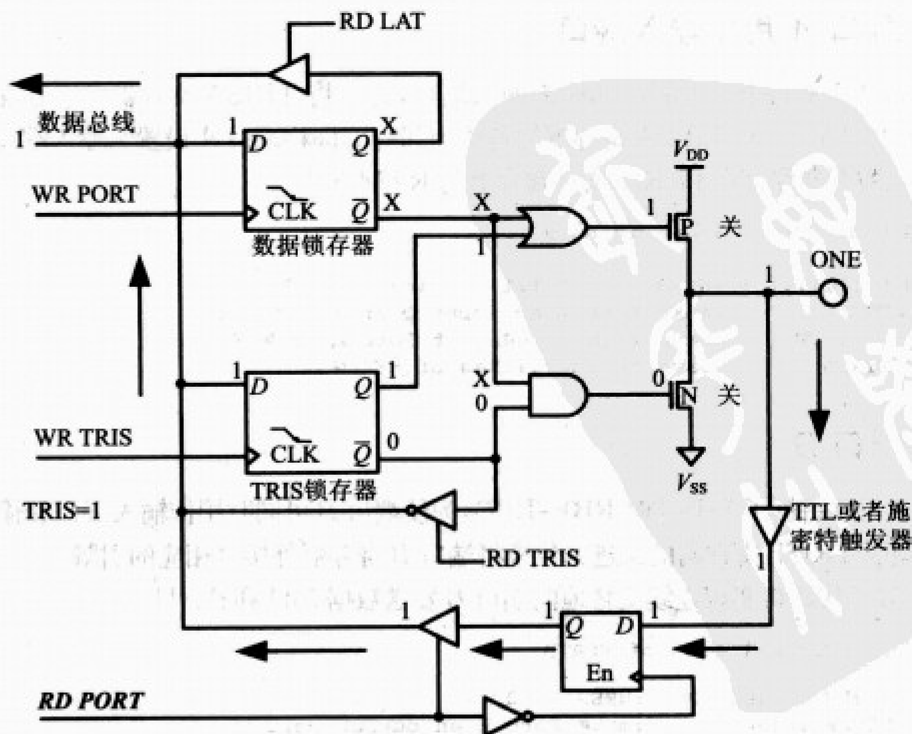


图 4-6 从 PIC18 单片机的一个引脚输入(读)1

4.1.4 端口 A

端口 A 总共占据了 7 个引脚(RA0～RA6),但对于 PIC18F458 来说,引脚 A6 被用作 OSC2 引脚。如果用晶体振荡器来为 PIC18 芯片提供时钟频率,那么端口 A6 将不可用,这一点将在第 8 章讲述。

为使 A 可同时用作输入口和输出口,必须通过使能 TRISA 寄存器的二进制位来将端口 A 寄存器外接至相应的引脚。例如,以下代码将实现连续交替地向端口 A 发送数据 55H 和 AAH。

```
        ;toggle all bits of PORTA

        MOVLW B'00000000' ;WREG = 00000000 (binary)
        MOVWF TRISA       ;make Port A an output port (0 for Out)
L1      MOVLW 0x55        ;WREG = 55h
        MOVWF PORTA       ;put 55h on Port A pins
        CALL  DELAY
        MOVLW 0xAA        ;WREG = AAh
        MOVWF PORTA       ;put AAh on Port A pins
        CALL  DELAY
        GOTO  L1
```

必须注意的是,将 55H(01010101)取反后就变成了 AAH(10101010)。尽管将 55H 和 AAH 连续发送给端口 A 时,端口 A 寄存器的所有 8 位都做了翻转,但是只有 6 个引脚(RA0～RA5)会输出翻转后的数据。

135

4.1.5 端口 A 用作输入端口

为使端口 A 的每个二进制位都作为输入端口,必须将 TRISA 寄存器的各位都置为 1。在以下的程序代码中,对 TRISA 寄存器的各位全部写 1,即将端口 A 设置为输入端口,然后从端口 A 接收数据并保存在 fileReg 文件寄存器的 RAM 区中。

```
MYREG EQU   0X20  ;save it here

MOVLW B'11111111' ;WREG = 11111111 (binary)
MOVWF TRISA       ;make Port A an input port (1 for In)
MOVF  PORTA,W     ;move from fileReg of Port A to WREG
MOVWF MYREG       ;save it in fileReg of MYREG
```

4.1.6 端口 B

端口 B 总共占据了 8 个引脚(RB0～RB7),为使端口 B 可同时用作输入端口和输出端口,必须通过使能 TRISB 寄存器的二进制位来将端口 B 寄存器外接至相应的引脚。

例如,以下代码将实现连续交替地向端口 B 发送数据 55H 和 AAH。

```
        ;toggle all bits of PORTB

        MOVLW B'00000000' ;WREG = 00
        MOVWF TRISB       ;make Port B an output port
L1      MOVLW 0x55        ;WREG = 55h
        MOVWF PORTB       ;put 55h on port B pins
        CALL  DELAY
```

```
MOVLW 0xAA          ;WREG = AAh
MOVWF PORTB         ;put AAh on port B pins
CALL  DELAY
GOTO  L1
```

4.1.7 端口 B 用作输入端口

为使端口 B 的每个二进制位都作为输入端口，必须将 TRISB 寄存器的各位都置为 1。在以下的程序代码中，对 TRISB 寄存器的各位全部写 1，即将端口 B 设置为输入端口，然后从端口 B 接收数据并保存在 fileReg 文件寄存器的 RAM 区中。

```
MYREG EQU   0X25  ;save it here

MOVLW B'11111111' ;WREG = 11111111 (binary)
MOVWF TRISB       ;make Port B an input port (1 for In)
MOVF  PORTB,W     ;move from fileReg of Port B to WREG
MOVWF MYREG       ;save it in fileReg
```

136

4.1.8 端口 A 和端口 B 的双重功能

PIC18 通过端口 A 复用了一个 A/D 转换器来节省 I/O 引脚。表 4-3 给出了端口 A 引脚的第二(或更多)功能。关于如何使用端口 A 的 ADC 转换器，请参阅第 13 章。因为许多项目都会用到 ADC 转换器，所以端口 A 一般不用作简单的 I/O 功能。PIC18 还通过端口 B 复用了一些其他的功能来节省引脚。表 4-4 列出了端口 B 引脚的第二(或更多)功能。在以后的章节中将会学习如何使用端口 B 引脚的第二功能。

表 4-3 端口 A 的复用功能

位	功　能
RA0	AN0/CVREF
RA1	AN1
RA2	AN2/VREF−
RA3	AN3/VREF+
RA4	T0CKI
RA5	AN4/SS/LVDIN
RA6	OSC2/CLKO

表 4-4 端口 B 的复用功能

位	功　能
RB0	INT0
RB1	INT1
RB2	INT2/CANTX
RB3	CANRX
RB4	
RB5	PGM
RB6	PGC
RB7	PGD

4.1.9 端口 C

端口 C 总共占据了 8 个引脚(RC0～RC7)，为使端口 C 可同时用作输入端口和输出端口，必须通过使能 TRISC 寄存器的二进制位来将端口 C 寄存器外接至相应的引脚。例如，以下代码将实现连续交替地向端口 C 发送数据 55H 和 AAH。

```
        ;toggle all bits of PORTB

        MOVLW B'00000000'  ;WREG = 00
        MOVWF TRISC        ;make Port C an output port
L1      MOVLW 0x55         ;WREG = 55h
        MOVWF PORTC        ;put 55h on Port C pins
```

```
CALL  DELAY
MOVLW 0xAA          ;WREG = AAh
MOVWF PORTC         ;put AAh on Port C pins
CALL  DELAY
GOTO  L1
```

137

4.1.10 端口 C 用作输入端口

为使端口C的每个二进制位都作为输入端口,必须将TRISC寄存器的各位都置为1。在以下的程序代码中,对TRISC寄存器的各位全部写1,即将端口C设置为输入端口,然后从端口C接收数据并保存在fileReg文件寄存器的RAM区中。

```
MYREG EQU 0x20      ;save it here
MOVLW B'11111111'   ;WREG = 11111111 (binary)
MOVWF TRISC         ;make Port C an input port (1 for In)
MOVF  PORTC,W       ;move from fileReg of Port C to WREG
MOVWF MYREG         ;save it in fileReg
```

4.1.11 端口 D

端口D总共占据了8个引脚(RD0~RD7),为使端口D可同时用作输入端口和输出端口,必须通过使能TRISD寄存器的二进制位来将端口D寄存器外接至相应的引脚。例如,以下代码将实现连续交替地向端口D发送数据55H和AAH。

```
      ;toggle all bits of PORTD

      CLRF  TRISD        ;make Port D an output port
L1    MOVLW 0x55         ;WREG = 55h
      MOVWF PORTD        ;put 55h on Port D pins
      CALL  DELAY
      MOVLW 0xAA         ;WREG = AAh
      MOVWF PORTD        ;put AAh on Port D pins
      CALL  DELAY
      BRA   L1           ;we can use GOTO
```

4.1.12 端口 D 作为输入端口

为使端口D的每个二进制位都作为输入端口,必须将TRISD寄存器的各位都置为1。在以下的程序代码中,对TRISD寄存器的各位全部写1,即将端口D设置为输入端口,然后从端口D接收数据并保存在fileReg文件寄存器的RAM区中。

```
MYREG EQU 0x20      ;save it here
SETF  TRISD         ;TRISD = 11111111 (binary) = PORTD = Input
MOVF  PORTD,W       ;move from Port D to WREG
MOVWF MYREG         ;save it in fileReg
```

138

4.1.13 端口 C 和端口 D 的双重功能

端口C的第二功能如表4-5所示。关于如何使用端口C的第二功能将在后面的章节介绍。端口C的第二功能如表4-6所示。关于端口D的第二功能的使用,也将在后面的章节中介绍。

表 4-5 端口 C 的复用功能

位	功 能
RC0	T1OSO/T1CKI
RC1	T1OSI
RC2	CCP1
RC3	SCK/SCL
RC4	SDI/SDA
RC5	SDO
RC6	TX/CK
RC7	RX/DT

表 4-6 端口 D 的复用功能

位	功 能
RD0	PSP0/C1IN+
RD1	PSP1/C1IN−
RD2	PSP2/C2IN+
RD3	PSP3/C2IN−
RD4	PSP4/ECCP1/P1A
RD5	PSP5/P1B
RD6	PSP6/P1C
RD7	PSP7/P1D

4.1.14 端口 *E*

在 PIC18F458/4580 中，端口 *E* 总共占据了 3 个引脚(RE0～RE2)。端口 *E* 用于 3 个附加的模拟输入或者简单 I/O 端口，即 AN5、AN6 和 AN7。同其他的端口一样，端口 *E* 也有自己的第二功能。关于端口 *E* 的使用将在以后的章节中学习。

4.1.15 访问 8 位数据的不同方法

正如前面学习过的许多例子，以下的程序代码将实现对端口 *B* 的 8 位数据的访问。

```
        ;toggle all bits of PORTB

        MOVLW 0x0           ;WREG = 00
        MOVWF TRISB         ;make Port B an output port
L1      MOVLW 0x55          ;WREG = 55h
        MOVWF PORTB         ;put 55h on Port B pins
        CALL  DELAY
        MOVLW 0xAA          ;WREG = AAh
        MOVWF PORTB         ;put AAh on Port B pins
        CALL  DELAY
        GOTO  L1
```


上面的程序代码是连续地翻转端口 *B* 的所有二进制位。该程序代码的另一种写法如下。

```
        CLRF  TRISB         ;make Port B an output port
L1      MOVLW 0x55          ;WREG = 55h
        MOVWF PORTB         ;put 55h on Port B pins
        CALL  DELAY
        MOVLW 0xAA          ;WREG = AAh
        MOVWF PORTB         ;put AAh on Port B pins
        CALL  DELAY
        GOTO  L1
```

下面的程序代码同样可以实现上述的端口 *B* 的 8 位数据访问。这是一种被称为读-修改-写的技术。

```
CLRF  TRISB         ;make Port B an output port
MOVLW 0x55          ;WREG = 55h
MOVWF PORTB         ;put 55h on Port B pins
```

```
L2      COMF   PORTB, F     ;toggle bits of Port B
        CALL   DELAY
        BRA    L2
```

4.1.16　读取后紧接的写 I/O 操作

由于存在时序问题,应当避免连续的两个 I/O 操作。重写并考察前面的程序代码,从端口 *C* 读取数据并写入端口 *B*。

```
        CLRF   TRISB ;clear TRISB to make PORTB an output port
        SETF   TRISC        ;set TRISC all 1s (Port C as Input)
L4      MOVF   PORTC,W      ;get data from Port C into WREG
        NOP                 ;NEED some NOP to ensure data is in WREG
        MOVWF  PORTB        ;before it is sent to Port B
        BRA    L4           ;keep doing it
```

这里,通过插入 NOP 指令(或者其他指令)来保证数据在输出到端口 *B* 之前已被写入 WREG。在 CPU 设计中,这就是所谓的*数据依赖*。这种数据依赖通常是指 RAW(读-后-写),NOP 指令的引入在流水线中插入了一个间隔操作,从而可以消除由 RAW 引起的数据依赖,如图 4-7所示。避免该问题发生的一个方法是使用 4 B 的 MOVFF 指令,其程序代码如下。


```
        CLRF   TRISB        ;make Port B an output port
        SETF   TRISC        ;TRISC = FFh (Port C Input)
L5      MOVFF  PORTC,PORTB  ;get from Port C and send to PORTB
        BRA    L5           ;keep doing it
```

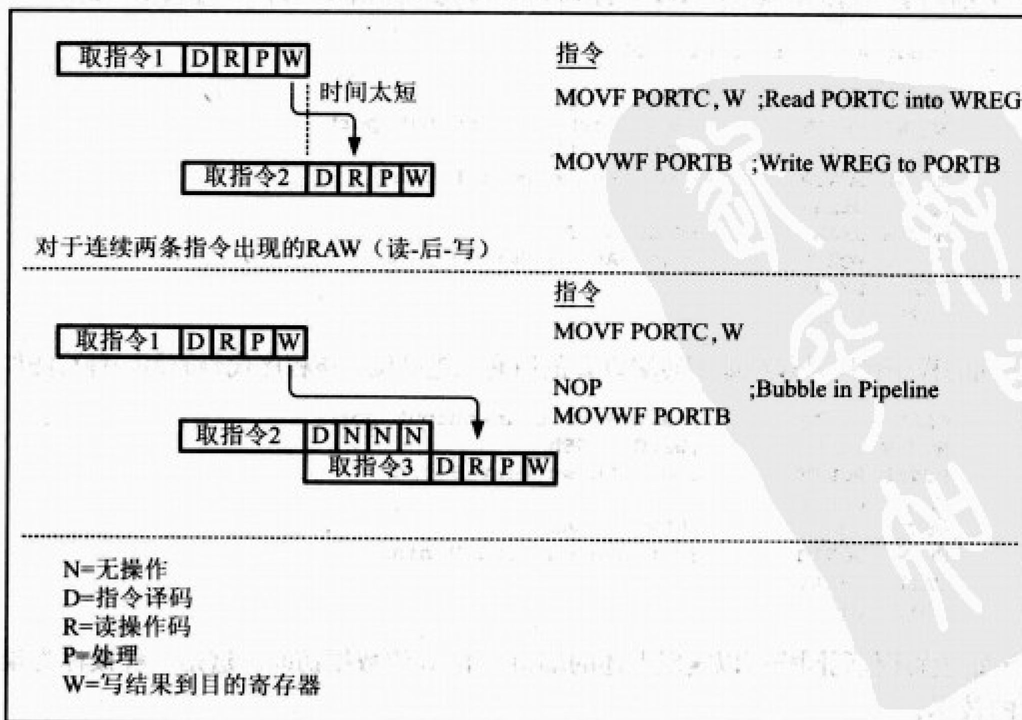


图 4-7　RAW 的流水线结构

4.1.17 复位时的端口状态

在系统复位时，所有端口的TRIS寄存器值均为FFH，如表4-7所示，即在复位时它们都被设置为输入端口。

表4-7 PIC18单片机TRIS寄存器的复位值

寄存器	复位值(二进制)	寄存器	复位值(二进制)
TRISA	11111111	TRISC	11111111
TRISB	11111111	TRISD	11111111

注意：系统复位时所有的端口均为输入端口。

4.1.18 复习题

1. PIC18F458总共有________个端口。
2. 判断对错：PIC18F458的所有端口都是8个引脚。
3. 试列举PIC18F458含有8个引脚的所有端口。
4. 判断对错：当系统复位时，所有的I/O端口都被配置为输出端口。
5. 编制程序，将数据99H传送给端口 *B* 和端口 *C*。
6. 为把端口 *B* 设置为输出端口，需要将数据________写到寄存器________中。
7. 为把端口 *B* 设置为输入端口，需要将数据________写到寄存器________中。

例4-1 为PIC18编写测试程序，每隔1/4 s将PORTB、PORTC和PORTD的各位翻转一次。假设晶振频率为4 MHz。

解：

```
;tested with MPLAB for the PIC18F458 and XTAL = 4 MHz

        list P=PIC18F458
#include P18F458.INC

R1 equ 0x07
R2 equ 0x08

ORG 0
        CLRF   TRISB          ;make Port B an output port
        CLRF   TRISC          ;make Port C an output port
        CLRF   TRISD          ;make Port D an output port
        MOVLW  0x55           ;WREG = 55h
        MOVWF  PORTB          ;put 55h on Port B pins
        MOVWF  PORTC          ;put 55h on Port C pins
        MOVWF  PORTD          ;put 55h on Port D pins
L3      COMF   PORTB,F        ;toggle bits of Port B
        COMF   PORTC,F        ;toggle bits of Port C
        COMF   PORTD,F        ;toggle bits of Port D
        CALL   QDELAY         ;quarter of a second delay
        BRA    L3

;-----------1/4 SECOND DELAY
QDELAY
```

```
        MOVLW  D'200'
        MOVWF  R1
D1      MOVLW  D'250'
        MOVWF  R2
D2      NOP
        NOP
        DECF  R2, F
        BNZ  D2
        DECF  R1, F
        BNZ  D1
        RETURN
        END
```

计算结果如下：

4 MHz/4=1 MHz

1/1 MHz=1 μs

延迟时间=250×200×5MC×1 μs=250 000 μs 。(如果计上程序头,那么延迟时间为250 800 μs。具体计算请参阅例 3-17。)

延迟时间的大小可使用 MPLAB 仿真器来验证。

4.2 I/O 位操作编程

这一节将继续介绍 PIC18 的 I/O 指令。要特别关注的是 I/O 位操作,因为它是 PIC 系列芯片的一个强大而又广泛使用的特性。

4.2.1 I/O 端口与位寻址

有时候,人们只需要访问一两个二进制位,而不是整个 8 位。PIC 的 I/O 端口的 一个重要特征就是,它允许在不改变端口其他二进制位的前提下访问其中的一个或者几个二进制位。对于所有的 PIC 端口,既可以访问端口的所有 8 位,也可以访问其中的某几位而不改变其余的位。表 4-8 列出了 PIC18 的伪指令。虽然表 4-9 所列的指令可以用于数据 RAM 文件寄存器中的任何寄存器,但是在 I/O 端口操作中用得更多。在后面的全部章节中都将看到这些指令的使用。

表 4-8 PIC18 的位(面向位的)指令

指 令	功 能
BSF fileReg, bit	位设置 fileReg (置位:bit=1)
BCF fileReg, bit	位清除 fileReg (清零位:bit=0)
BTG fileReg, bit	位翻转 fileReg (取反,bit=/bit)
BTFSC fileReg, bit	位测试 fileReg,若清零则跳过(如果 bit=0,那么跳过下一条指令)
BTFSS fileReg, bit	位测试 fileReg,若置位则跳过(如果 bit=1,那么跳过下一条指令)

表 4-9 PIC18F458/4580 端口的位寻址能力

PORTA	PORTB	PORTC	PORTD	PORTE	端口位
RA0	RB0	RC0	RD0	RE0	D0
RA1	RB1	RC1	RD1	RE1	D1

(续)

PORTA	PORTB	PORTC	PORTD	PORTE	端口位
RA2	RB2	RC2	RD2	RE2	D2
RA3	RB3	RC3	RD3		D3
RA4	RB4	RC4	RD4		D4
RA5	RB5	RC5	RD5		D5
	RB6	RC6	RD6		D6
	RB7	RC7	RD7		D7

接下来讨论所有的位寻址指令和它们的使用方法。

4.2.2 BSF(置位 fileReg)

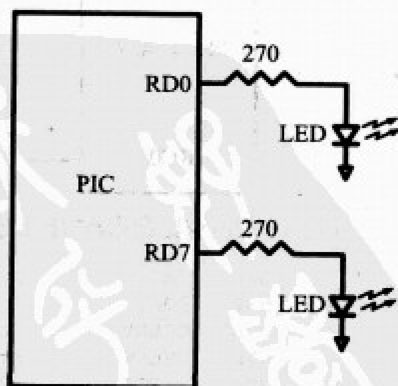
为了将给定文件寄存器的某个位设置为 1，常用的指令是 BSF fileReg, bit_num，其中，fileReg 可以是文件寄存器的任意位置，而 bit_num 是期望设置的位序号，可以是 0 到 7 中的任意数字。尽管面向位的指令可用于文件寄存器的二进制位(D0～D7)，可是它们在嵌入式系统中通常都用于 I/O 端口操作。例如，指令 BSF PORTB, 5 将端口 B 的第 5 位置为 1。

143

例 4-2 在端口 D 的每个引脚上都连接一个 LED 灯。编制程序按照 RD0 到 RD7 的顺序点亮 LED 灯。在点亮下一个 LED 等之前调用时延子例程。

解：

```
CLRF   TRISD        ;make PORTD an output port
BSF    PORTD,0      ;bit set turns on RD0
CALL   DELAY        ;delay before next one
BSF    PORTD,1      ;turn on RD1
CALL   DELAY        ;delay before next one
BSF    PORTD,2
CALL   DELAY
BSF    PORTD,3
CALL   DELAY
BSF    PORTD,4
CALL   DELAY
BSF    PORTD,5
CALL   DELAY
BSF    PORTD,6
CALL   DELAY
BSF    PORTD,7
CALL   DELAY
```



4.2.3 BCF(清零 fileReg)

为了将给定文件寄存器的某个位清零，常用的指令是 BCF fileReg, bit_num。要记住的是，若要引脚上反映 I/O 端口寄存器的变化，就必须激活 TRISx 寄存器相应的二进制位。例如，下面的程序代码实现连续地翻转引脚 RB2。

```
        BCF    TRISB, 2           ;bit = 0, make RB2 an output pin
AGAIN BSF    PORTB, 2           ;bit set (RB2 = high)
        CALL   DELAY
        BCF    PORTB, 2           ;bit clear(RB2 = low)
        CALL   DELAY
        BRA    AGAIN
```

144

例 4-3　分别编制程序，实现下面的功能：

(a) 在端口 C 的第 0 位输出一个占空比为 50%的方波；

(b) 在端口 C 的第 3 位输出一个占空比为 66%的方波。

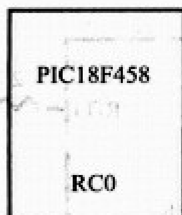
解：

(a) 50%的占空比意味着开、关状态或者脉冲的高低电平具有相同的持续时间。因此，可以以一定的时延间隔来翻转 RC0。

```
       BCF   TRISC,0      ;clear TRIS bit for RC0 = out
HERE   BSF   PORTC,0      ;set to HIGH RC0 (RC0 = 1)
       CALL  DELAY        ;call the delay subroutine
       BCF   PORTC,0      ;RC0 = 0
       CALL  DELAY
       BRA   HERE         ;keep doing it
```

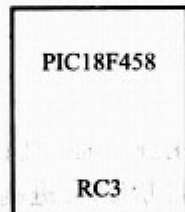
以上程序代码的另一种写法是：

```
       BCF   TRISC,0      ;make RC0 = out
HERE   BTG   PORTC,0      ;complement bit 0 of PORTC
       CALL  DELAY        ;call the delay subroutine
       BRA   HERE         ;keep doing it
```



(b) 66%的占空比意味着“开”状态的持续时间是“关”状态持续时间的两倍。

```
       BCF   TRISC,3      ;clear TRISC3 bit for output
BACK   BSF   PORTC,3      ;RC3 = 1
       CALL  DELAY        ;call the delay subroutine
       CALL  DELAY        ;twice for 66%
       BCF   PORTC,3      ;RC3 = 0
       CALL  DELAY        ;call delay once for 33%
       BRA   BACK         ;keep doing it
```



145

4.2.4 BTG（位翻转 fileReg）

为了将给定文件寄存器的某个位翻转，常用的指令是 BTG fileReg, bit_num。

```
        BCF    TRISB, 2              ;make RB2 an output pin
BACK    BTG    PORTB, 2              ;toggle pin RB2 only
        CALL   DELAY
        BRA    BACK
```

注意，RB2 是端口 *B* 的第 3 个二进制位(第 1 个二进位是 RB0，第 2 个二进制位是 RB1，以此类推)，如表 4-9 所示。例 4-2 给出了 I/O 端口位操作的例子。

注意，例 4-2 不会对其他未使用的 I/O 端口二进制位造成任何影响。I/O 端口位寻址是 PIC 微控制器的最重要特性之一，也是许多设计人员选用 PIC 芯片的原因之一。关于 I/O 端口的位寻址能力的应用将在后面的章节中陆续介绍。

4.2.5 检测输入引脚

若要根据文件寄存器中某个位的状态而做出决策，通常需要使用指令 BTFSC(位测试 fileReg，若为零则跳过)和指令 BTFSS(位测试 fileReg，若为 1 则跳过)。这些位指令在 I/O 端口操作中的运用十分广泛，使用它们可以监视某个引脚，并根据其状态(0 或者 1)来做出判断。此外，应注意的是，BTFSC 指令和 BTFSS 指令还可用于文件寄存器的任何一个二进制位，包括 I/O 端口 *A*、*B*、*C*、*D* 等。

4.2.6 BTFSS（位测试 fileReg，若为 1 则跳过）

要监测某个位的状态是否为高电平(HIGH)，可以使用指令 BTFSS。该指令测试二进制位，若该位为高电平则跳过下一条指令。关于 BTFSS 指令的使用，请参阅例 4-4。

1010	bbba	ffff	ffff

$0 \leqslant f \leqslant FF$

$0 \leqslant b \leqslant 7$

4.2.7 BTFSC（位测试 fileReg，若为 0 则跳过）

要监测某个位的状态是否为低电平(LOW)，可以使用指令 BTFSC。该指令测试二进制位，若该位为低电平则跳过下一条指令。关于 BTFSS 指令的使用，请参阅例 4-5。

1011	bbba	ffff	ffff

$0 \leqslant f \leqslant FF$

$0 \leqslant b \leqslant 7$

例 4-4 编制程序，实现下面的功能：

(a) 监测 RB2 引脚，直到它变成高电平；

(b) 当 RB2 引脚变为高电平时，向端口 C 写数据 45H，并让 RD3 引脚输出一个由高到低的脉冲信号。

解：

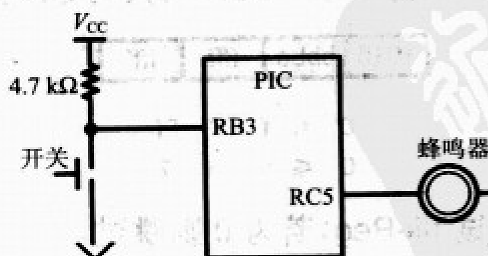
```
      BSF    TRISB,2      ;make RB2 an input
      CLRF   TRISC        ;make PORTC an output port
      BCF    PORTD,3      ;make RD3 an output
      MOVLW  0x45         ;WREG = 45h
AGAIN BTFSS  PORTB,2      ;bit test RB2 for HIGH
      BRA    AGAIN        ;keep checking if LOW
      MOVWF  PORTC        ;issue WREG to Port C
      BSF    PORTD,3      ;bit set fileReg RD3 (H-to-L)
      BCF    PORTD,3      ;bit clear fileReg RD3 (L)
```

在上面的程序中，只要 RB2 引脚维持低电平，则 BTFSS PORTB,2 将一直循环执行。当 RB2 引脚变为高电平时，将跳过分支转移指令而跳出循环，并向端口 C 写入数值 45H。然后，向 RD3 引脚输出一个由高到低的脉冲信号。

例 4-5 假设 RB3 引脚是一个输入端口，表示门报警器的状态。如果该引脚为低电平，那么表明门是开门状态。不断地监测该引脚。无论何时该引脚变为低电平，都要向 RC5 引脚发送一个由高到低的脉冲信号，打开蜂鸣器。

解：

```
      BSF    TRISB,3      ;make RB3 an input
      BCF    TRISC,5      ;make RC5 an output
HERE  BTFSC  PORTB,3      ;keep monitoring RB3 for HIGH
      BRA    HERE         ;stay in the loop
      BSF    PORTC,5      ;make RC5 HIGH
      BCF    PORTC,5      ;make RC5 LOW for H-to-L
      BRA    HERE
```



147

4.2.8 监测二进制位

人们通常也用位测试指令来检测某个二进制位的状态，从而做出执行下一步操作的决定。其具体的用法请参阅例 4-6 和例 4-7。

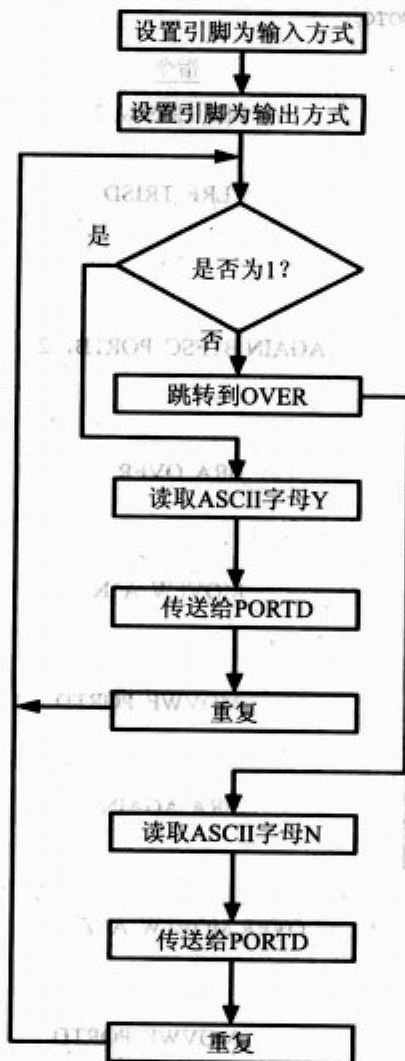
例 4-6 一个开关连接在 RB2 引脚上。编制程序，检查开关 SW 的状态，并执行下面的功能。

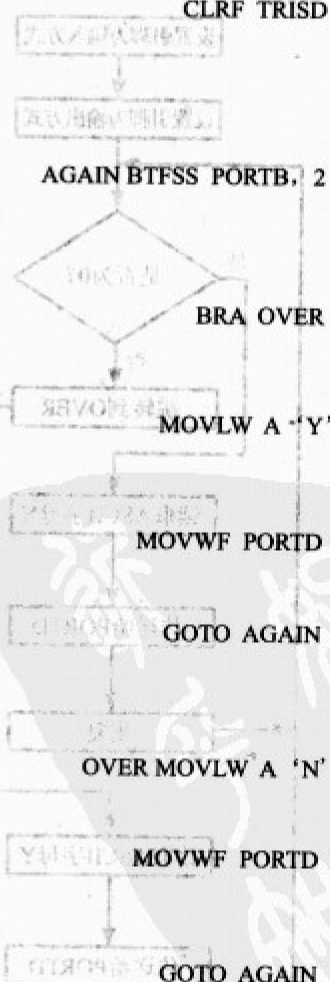
(a) 如果 SW=0，那么向 PORTD 发送字母 N。

(b) 如果 SW=1，那么向 PORTD 发送字母 Y。

解:

```
        BSF    TRISB,2      ;make RB2 an input
        CLRF   TRISD        ;make PORTD an output port
AGAIN   BTFSS  PORTB,2      ;bit test RB2 for HIGH
        BRA    OVER         ;it must be LOW
        MOVLW  A'Y'         ;WREG = 'Y' ASCII letter Y
        MOVWF  PORTD        ;issue WREG to PORTD
        GOTO   AGAIN        ;we can use BRA too
OVER    MOVLW  A'N'         ;WREG = 'N' ASCII  letter N
        MOVWF  PORTD        ;issue WREG to PORTD
        GOTO   AGAIN        ;we can use BRA too
```





例 4-7 一个开关连接在 RB2 引脚上。编制程序,检查开关 SW 的状态,并执行下面的功能。

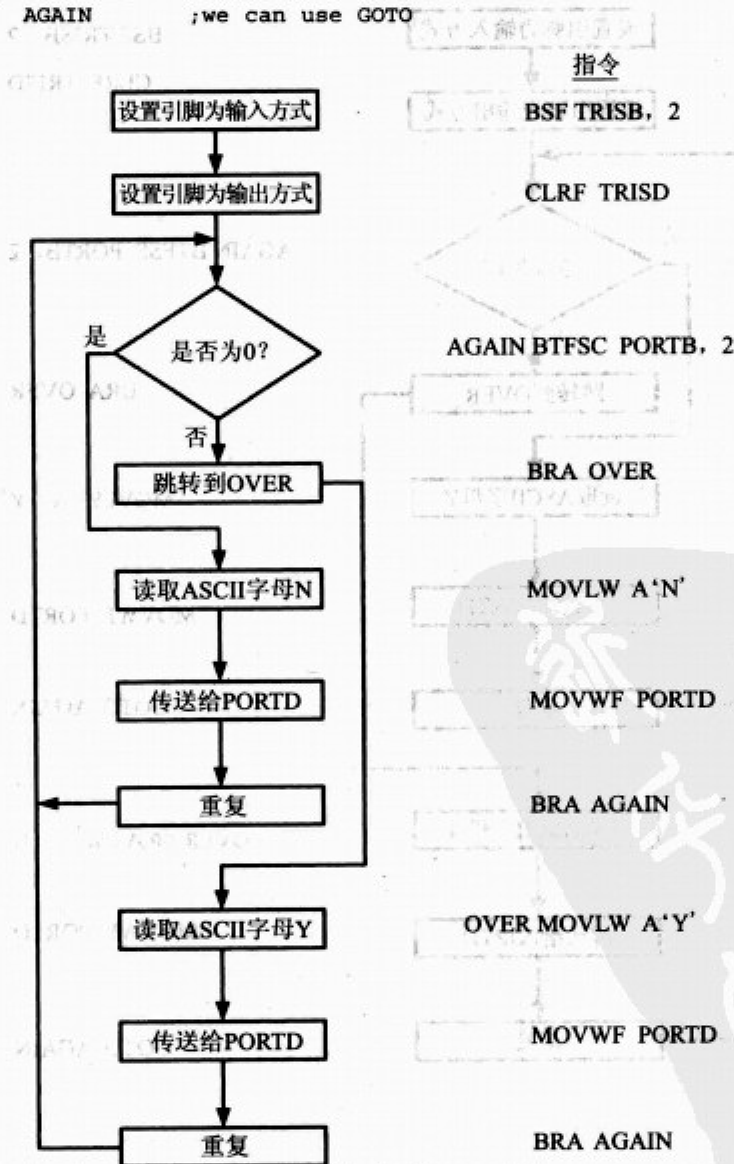
(a) 如果 SW=0,那么向 PORTD 发送字母 N。

(b) 如果 SW=1,那么向 PORTD 发送字母 Y。

使用指令 BTFSC 来检查 SW 的状态。下面的程序代码是例 4-6 的另外一种实现，即使用 BTFSC 指令代替了 BTFSS 指令。

解：

```
        BSF   TRISB,2      ;make RB2 an input
        CLRF  TRISD        ;make PORTD an output port
AGAIN   BTFSC PORTB,2      ;bit test RB2 for LOW
        BRA   OVER         ;it must be HIGH
        MOVLW A'N'         ;WREG = 'N' ASCII letter N
        MOVWF PORTD        ;issue WREG to PORTD
        BRA   AGAIN        ;we can use GOTO
OVER    MOVLW A'Y'         ;WREG = 'Y' ASCII letter Y
        MOVWF PORTD        ;issue WREG to PORTD
        BRA   AGAIN        ;we can use GOTO
```



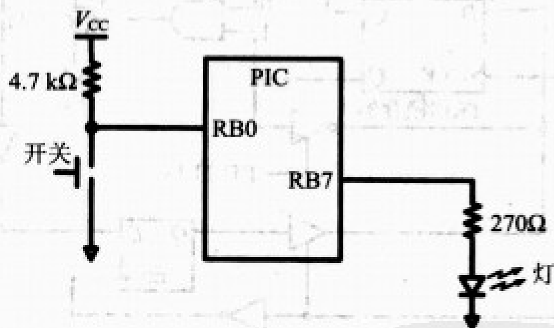
4.2.9 读取二进制位

人们通常也用位测试指令来读取某个二进制位的状态，并保存或者将它传送给其他位。其具体的用法请参阅例 4-8 和例 4-9。

例 4-8 一个开关连接在 RB0 引脚上，一个 LED 灯连接到 RB7 引脚。编制程序，读得 SW 的状态并将它传送给 LED 灯。

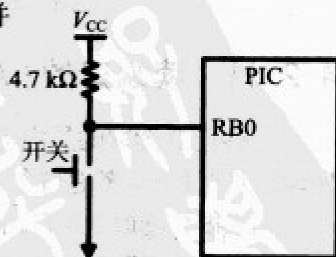
解：

```
        BSF    TRISB,0      ;make RB0 an input
        BCF    TRISB,7      ;make RB7 an output
AGAIN   BTFSS  PORTB,0      ;bit test RB0 for HIGH
        GOTO   OVER         ;it must be LOW (BRA is OK too)
        BSF    PORTB,7
        GOTO   AGAIN        ;we can use BRA too
OVER    BCF    PORTB,7
        GOTO   AGAIN        ;we can use BRA too
```



例 4-9 一个开关连接在 RB0 引脚上。编制程序，读得 SW 的状态并将它保存在 fileReg 位置 0x20 的 D0 位。

解：




```
MYBITREG EQU 0x20 ;set aside loc 0x20 reg

      BSF    TRISB,0       ;make RB0 an input
AGAIN BTFSS  PORTB,0       ;bit test RB0 for HIGH
      GOTO   OVER          ;it must be LOW (BRA is OK too)
      BSF    MYBITREG,0    ;set bit 0 of fileReg
      GOTO   AGAIN         ;we can use BRA too
OVER  BCF    MYBITREG,0    ;clear bit 0 of fileReg
      GOTO   AGAIN         ;we can use BRA too
```


4.2.10 读输入引脚与读 LATx 端口

在读端口时，一些指令是读端口引脚的状态，而其他一些指令则是读叫作 LATx 的内部端口锁存器的状态。因此，当读端口时就存在下面的 2 种可能：

(1) 读输入引脚的状态；

(2) 读 LAT 寄存器的内部锁存器。

用于读端口的两类指令必须加以区别，因为在 PIC 编程中就是因为经常混淆二者而出错，在考虑外部硬件时这种错误尤其突出。这里将简短地讨论这两类指令。然而，读者有必须去学习和理解关于读端口的文献，以及附录 C.2 给出的端口内部工作原理的资料。图 4-8 再次给出了端口的结构图。除 PORTx 寄存器和 TRISx 寄存器外，LATx 寄存器是与 PIC18 有关的第三个重要的寄存器。

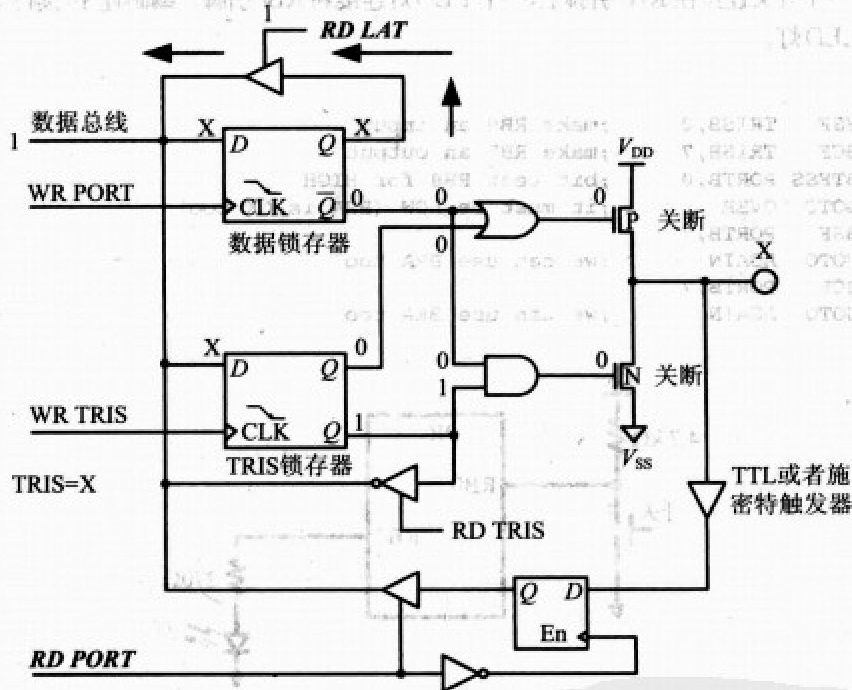


图 4-8 LATx 寄存器在读端口或者锁存器中的作用

4.2.11 读端口的 LATx

一些指令读取的是内部端口锁存器的内容，而不是外部引脚的状态。这些指令如表 4-10 所示。举个例子，考虑指令 COMF PORTB。当执行这个指令时，其动作顺序如下。

(1) 指令读取 LATB 的内部锁存器，并将数据传送给 CPU。

(2) 将数据取反。

(3) 将结果写回到 LATB 锁存器。

(4) 只有当 TRISB 位被清零时，引脚上的数据才会变化。

一般很少使用指令(如 COMF LATB,F)来读取锁存寄存器的内容的，尽管它是一条合法的指令。

从以上的讨论可以得出结论，读端口锁存器的指令一般是先读锁存器的值，执行某个操作，再将结果写回到端口锁存器。这个过程被称作读-修改-写。在进行该操作前，必须将端口设置为输出方式。

表 4-10 部分读-修改-写指令

指	令	功 能
ADDWF	fileReg, d	将 f 加到 WREG 中
BSF	fileReg, bit	位设置 fileReg（置位：bit=1）
BCF	fileReg, bit	位清零 fileReg（清零位，bit=0）
COMF	fileReg, d	将 f 取反
INCF	fileReg, d	将 f 加 1
SUBWF	fileReg, d	从 f 中减去 WREG 的内容
XORWF	fileReg, d	将 f 与 WREG 的内容执行异或操作

4.2.12 复习题

1. 判断对错：指令 BSF PORTB,1 将 RB1 引脚置为高电平而保持其他引脚的状态不变(如果 TRISB 的第 1 位被配置为输出)。
2. 给出一种方法，使用 PIC 指令连续地翻转 RB7 引脚。
3. 使用指令 BTFSS PORTC,5 的一个假设前提是 RC5 位是一个________(输入，输出)引脚。
4. 编制程序，读取 RB2 的状态，并将它输出至 RB0。
5. 编制程序，连续地翻转 RD7 和 RD0 位。

> **注 意**
>
> 强烈建议读者在连接外部硬件到 PIC 系统之前学习 C.2 节(附录 C)。错误地使用指令或者不能正确地连接端口引脚都将损坏 PIC 芯片的引脚。

152

小结

这章主要介绍了 PIC 的 I/O 端口。讨论了 PIC18F458 的 5 个端口：PORTA、PORTB、PORTC、PORTD 和 PORTE。这些端口都可以用作输入端口或者输出端口。所有的端口都具有第二功能。每个端口都有 3 个与之相关的寄存器：PORTx、TRISx 和 LATx，它们在 I/O 操作中的作用在本章也有讨论。本章还解释了 PIC 的 I/O 指令，并给出了大量的例子。本章最后讨论了 PIC 端口的位寻址能力。

习题

1. PIC18F458 具有________个引脚的 DIP 封装。
2. 在 PIC18F458 中，有多少个引脚是分配给 V_{CC} 和 GND?
3. 在 PIC18F458 中，有多少个引脚是被指定为 I/O 端口引脚?
4. 在 40 引脚 DIP 封装中，有多少个引脚被指定为 PORTA? 它们的引脚编号分别是多少呢?
5. 在 40 引脚 DIP 封装中，有多少个引脚被指定为 PORTB? 它们的引脚编号分别是多少呢?
6. 在 40 引脚 DIP 封装中，有多少个引脚被指定为 PORTC? 它们的引脚编号分别是多少呢?
7. 在 40 引脚 DIP 封装中，有多少个引脚被指定为 PORTD? 它们的引脚编号分别是多少呢?

8. 当系统复位时,端口的所有位被设置为________(输入,输出)。
9. 对于PIC18,为了使用简单的I/O端口,必须对哪个寄存器编程呢?
10. 试解释TRISx寄存器和PORTx寄存器在I/O操作中的作用。
11. 编制程序,从端口C读取8位数据并传送给端口B和端口D。
12. 编制程序,从端口D读取8位数据并传送给端口B和端口C。
13. 哪些引脚是用于RxD和TxD的?
14. 给出分配给端口A~C和它们的寄存器TRIS的、在文件寄存器中的RAM数据地址。
15. 编制程序,连续地翻转PORTB和PORTC的所有二进制位:
    (a) 使用数据AAH和55H; (b) 使用COMF指令。
16. PIC18的哪些端口是有位寻址能力的?
17. 对于PIC端口,位寻址的优势在哪里?
18. 当RB2作为单独的位被访问时,这称为________。
19. COMF PORTB是合法的指令吗?

153

20. 编制程序,连续地翻转RB2和RB5而不影响其他的二进制位。
21. 编制程序,连续地翻转RD3、RD7和RC5而不影响其他的二进制位。
22. 编制程序,监测位RC3。当RC3为高电平时,向PORTD发送55H。
23. 编制程序,监测位RB7。当RB7为低电平时,连续地向PORTC发送55H和AAH。
24. 编制程序,监测位RE0。当RE0为高电平时,向PORTB发送99H。当RE0为低电平时,向PORTC发送66H。
25. 编制程序,监测位RB5。当RB5为高电平时,向RB3引脚发送从低到高再到低的脉冲信号。
26. 编制程序,读取RC3位的状态,并将它输出到RC4。
27. RB4指的是PORTB的哪一位?
28. 创建流程图,编制程序,读取RD7和RD6的状态,并将它们分别输出到RC0和RC7。

复习题答案

4.1 节

1. 5　　2. 错误。　　3. PORTB、PORTC和PORTD　　4. 错误。

5.
```
MOVLW    0x99
MOVWF    PORTB
MOVWF    PORTC
```

6. 00,TRISB　　7. FFH,TRISB

4.2 节

1. 正确。

2.
```
   BCF TRISB,7
H1 BTG PORTB,7
   BRA H1
```

3. 输入

4.
```
         BSF    TRISB,2
         BCF    TRISB,0
AGAIN    BTFSS  PORTB,2
         BRA    OVER
         BSF    PORTB,0
         BRA    AGAIN
OVER     BCF    PORTB,0
         BRA    AGAIN
```

5.

```
        BCF   TRISD,0
        BCF   TRISD,7
H2      BTG   PORTD,0
        BTG   PORTD,7
        BRA   H2
```

第5章 算术、逻辑指令和程序示例

学习目标：

- ☐ 定义 PIC 无符号数的取值范围
- ☐ 无符号数的加减法指令编程
- ☐ BCD 数的加法运算
- ☐ PIC 无符号数乘法指令编程
- ☐ PIC 除法程序的编制
- ☐ PIC 汇编指令 AND、OR 和 EX-OR 的编程
- ☐ 使用 PIC 逻辑指令进行位操作
- ☐ 使用比较和跳转指令控制程序
- ☐ PIC 移位指令和数据串行化的编程
- ☐ 阐述数据表示的 BCD 码(用二进制编码的十进制)系统
- ☐ 压缩和非压缩 BCD 码的对比
- ☐ ASCII 码和 BCD 码数据转换的程序编制

本章将讨论所有的 PIC 算术和逻辑指令。给出的示例程序用来说明这些指令的应用。5.1 节将介绍关于无符号数的加法、减法、乘法和除法的指令和程序。5.2 节将讨论有符号数的相关运算问题。5.3 节将探讨逻辑指令 AND、OR、XOR 和 COMPARE 指令。移位指令和数据串行化将在 5.4 节中讨论。5.5 节将提供一些实用的程序，如 BCD 和 ASCII 码的转换程序。

5.1 算术指令

无符号数被定义成这样一种数据，即所有的二进制位都被用来表示数据，而没有用于表示正号或者负号的位。也就是说，操作数是位于 00～FFH(或者十进制的0～255)的 8 位数据。

5.1.1 无符号数的加法

在 PIC 中，要实现数值的加法就必须使用寄存器 WREG。加法指令的一种形式是：

```
ADDLW   K   ;WREG = WREG + K
```

最后的和被保存在 WREG 寄存器中。这条指令可能会改变状态寄存器中的标志位 C、DC、Z、N 或者 OV，具体情况取决于运算结果。ADDLW 指令对 N 和 OV 标志位的影响将在 5.3 节中讨论，因为这两个标志位主要与有符号数的运算相关。见例 5-1。

例 5-1 指出执行完以下的指令后，标志寄存器的变化。

```
MOVLW 0xF5          ;WREG = F5 hex
ADDLW 0xB           ;WREG = F5 + 0B = 00 and C = 1
```

解：

```
    F5H          1111  0101
  + 0BH        + 0000  1011
   100H          0000  0000
```

在执行加法运算后，WREG 寄存器的值为 00，状态寄存器的内容如下：

C=1，因为 D7 位有进位；

Z=1，因为 WREG 中的结果为 0；

DC=1，因为从 D3 到 D4 有进位。

5.1.2 ADDWF 和单字节的加法

指令 ADDWF fileReg,d 的功能是将 WREG 中的内容和存储在文件寄存器 RAM 地址中的单字节数相加。注意，这里必须用到 WREG 寄存器，因为在 PIC 汇编语言中，从存储器到存储器的算术运算操作是不允许的。为了求取任意多个操作数的和，必须在执行完每两个操作数的加法后检查进位标志位。例 5-2 说明了这一过程，在操作数被加到 WREG 时，文件寄存器的地址 7 被用来累计进位。在第 6 章中，我们将用循环的形式来改写这段程序代码，以用于实现多个操作数的加法。

156

例 5-2 假设文件寄存器 RAM 地址 40～43H 存储了以下的十六进制数，编写程序，计算它们的代数和。在程序的末尾，文件寄存器的地址 6 用于存放所得结果的低字节位，地址 7 用于存放所得结果的高字节位。

40 = (7D)

41 = (EB)

42 = (C5)

43 = (5B)

解：

```
L_Byte EQU   0x6          ;assign RAM location 6 to L_byte of sum
H_Byte EQU   0x7          ;assign RAM location 7 to H_byte of sum

       MOVLW 0            ;clear WREG (WREG = 0)
       MOVWF H_Byte       ;H_Byte = 0
       ADDWF 0x40,W       ;WREG = 0 + 7DH = 7DH , C = 0
       BNC   N_1          ;branch if C = 0
       INCF  H_Byte,F     ;increment (now H_Byte = 0)
N_1    ADDWF 0x41,W       ;WREG = 7D + EB = 68H and C = 1
       BNC   N_2          ;
       INCF  H_Byte,F     ;C = 1, increment (now H_Byte = 1)
N_2    ADDWF 0x42,W       ;WREG = 68 + C5 = 2D and C = 1
       BNC   N_3          ;
       INCF  H_Byte       ;C = 1, increment (now H_Byte = 2)
N_3    ADDWF 0x43,W       ;WREG = 2D + 5B = 88H and C = 0
       BNC   N_4          ;
       INCF  H_Byte,F     ;(H_Byte = 2)
N_4    MOVWF L_Byte       ;now L_Byte = 88h
```

最后，由于 7D+EB+C5+5B=288H，所以有文件寄存器地址 6=(88)，文件寄存器地址7=(02)。使用寄存器间接寻址方式编制上面的程序代码具有更高的效率，详情请参阅第6章。

5.1.3 ADDWFC 和 16 位数的加法

当进行16位加法计算的时候，将会涉及从低字节向高字节的进位。为了区别于单字节数的加法，这种16位数的加法又被叫作*多字节加法*。指令 ADDWFC(ADDW 和带进位的 fileReg)就是在这些情况下使用的。

157

举个例子，3CE7H + 3B8DH 的加法可写成如下的格式：

```
    1
    3C   E7
+   3B   8D
    78   74
```

当第一个字节相加的时候，有进位产生(E7+8D=74，CY=1)。这个进位将被传递到高字节，结果就是 3C+3B+1=78(全部都是十六进制)。例5-3给出了实现上述加法的程序代码。

例5-3 编制程序，实现两个十六位数的相加。操作数是 3CE7H 和 3B8DH。假设文件寄存器地址 6=(8D)，地址 7=(3B)。请将所得的和存放在地址6和地址7，其中地址6为低字节。

解：

```
;location 6 = (8D)
;location 7 = (3B)

MOVLW 0xE7         ;load the low byte now (WREG = E7H)
ADDWF 0x6,F        ;F = W + F = E7 + 8D = 74 and CY = 1
MOVLW 0x3C         ;load the high byte (WREG = 3CH)
ADDWFC 0x7,F       ;F = W + F + carry, adding the upper byte
                   ;with Carry from lower byte
                   ;F = 3C + 3B + 1 = 78H (all in hex)
```

注意，指令 ADDWF 用于低字节加法，指令 ADDWFC 用于高字节加法。

5.1.4 BCD(二进制编码的十进制数)数字系统

BCD表示二进制编码的十进制数，因为人们平时习惯于使用0～9表示数字，而不是二进制或十六进制数，所以BCD数是需要的。二进制表示的0～9就被叫作BCD数，如图5-1所示。在计算机文献中，有两种表示BCD数的方法：非压缩的BCD数和压缩的BCD数。下面将分别阐述这两种BCD表示法。

十进制数	*BCD*
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

图5-1 BCD数

5.1.5 非压缩 BCD 数

在非压缩的BCD数中，字节的低4位用来表示BCD数，其余的位为0。例如，00001001 和 00000101 分别是9和5的非压缩BCD数。

非压缩 BCD 数的存储需要一个字节的存储空间或者一个 8 位寄存器。

5.1.6 压缩 BCD 数

在压缩的 BCD 数中,一个字节包括两个 BCD 数,一个位于低 4 位,一个位于高 4 位。例如,01011001 是 59H 的压缩 BCD 数。非压缩 BCD 操作数的存储只需要一个字节的内存空间。因此,使用压缩 BCD 数的存储效率是非压缩 BCD 数的存储效率的两倍。

158

在 BCD 数的加法中存在一个结果修正的问题。在两个压缩 BCD 数相加后,所得的结果不再是 BCD 数,例如:

```
MOVLW 0x17
ADDLW 0x28
```

将这两个数相加,得到 00111111B(3FH),这并不是一个 BCD 数! 一个 BCD 数只能有从 0001 到 1001(或者 0~9)的数字,也就是说,两个 BCD 数相加,所给出的结果必须是 BCD 数。以上加法运算的正确结果应是 17+28=45(01000101)。为了修正这个问题,程序员必须对低 4 位进行加 6 操作,即 3F+06=45H。高 4 位也存在同样的问题,高 4 位必须也加上 6(即 D9H+60H=139H),以保证结果是 BCD 数(52+87 =139)。这个问题十分普遍,大部分微处理器(如 PIC18)都有一个专门处理此问题的指令。在 PIC18 中,指令 DAW 被用来修正 BCD 数的加法结果。接下来将讨论指令 DAW。

5.1.7 DAW 指令

在 PIC18 中,DAW(decimal adjust WREG,WREG 的十进制调整)指令用来修正与 BCD 数相加有关的问题。DAW 指令只有一个操作数,即 WREG 的内容。若有必要,DAW 指令会对低位或者高位加 6,否则结果会保持不变。以下的例子说明了这一点。

```
MOVLW 0x47   ;WREG = 47H first BCD operand
ADDLW 0x25   ;hex(binary) addition (WREG = 6CH)
DAW          ;adjust for BCD addition (WREG = 72H)
```

当程序执行后,寄存器的内容将是 72H(47+25=72)。注意,DAW 指令只对 WREG 有效。

DAW 指令小结

在执行任何一条指令之后,

(1) 如果低 4 位大于 9,或者 DC=1,那么将低 4 位加上 0110;

(2) 如果高 4 位大于 9,或者 C=1,那么将高 4 位加上 0110。

在实际应用中,除了 BCD 加法和修正之外,就几乎没有用到 DC 标志位的地方。

```
MOVLW 0x00   ;WREG = 0
ADDLW 0x09   ;WREG = 0x09
ADDLW 0x08   ;WREG = 0x11, DC = 1
DAW          ;WREG = 0x17 (9 + 8 = 17)
```

作为另一个例子,下面将看到 57H+77H 的情况,所得结果是 CEH。如果将其当作 BCD 数的加法运算,那么这个结果显然是不对的。

159

```
十六进制        BCD
  57          0101 0111
+ 77        + 0111 0111
  CE          1100 1110
+ 66        + 0110 0110
 134        1 0011 0100   注意, C = 1
```

注意,与其他的处理器不同,PIC在执行DAW指令之前不需要使用算术指令。请看以下没有用到算术指令的例子。

```
MOVLW 0x0C   ;WREG = 00001100
DAW          ;WREG = 00001100 + 00000110 = 00010010 = 0x12
```

请继续学习例5-4。

例5-4 假设5个BCD数存储在以40H开头的RAM区域,如下所示。编制程序,计算所有数的和,要求结果是BCD数。

40=(71)

41=(88)

42=(69)

43=(97)

解:

```
L_Byte    EQU   0x6         ;assign RAM loc 6 to L_Byte of sum
H_Byte    EQU   0x7         ;assign RAM loc 7 to H_Byte of sum

      MOVLW     0           ;clear WREG (WREG = 0)
      MOVWF     H_Byte      ;H_Byte = 0
      ADDWF     0x40,W      ;WREG = 0 + 71H = 71H, C = 0
      DAW                   ;WREG = 71H
      BNC       N_1         ;branch if C = 0

      INCF      H_Byte,F    ;
N_1   ADDWF     0x41,W      ;WREG = 71 + 88 = F9H
      DAW                   ;WREG = 59H AND C = 1
      BNC       N_2         ;
      INCF      H_Byte,F    ;C = 1, increment (now H_Byte = 1)
N_2   ADDWF     0x42,W      ;WREG = 59 + 69 = C2 and Carry = 0
      DAW                   ;WREG = 28 and C = 1
      BNC       N_3         ;
      INCF      H_Byte      ;C = 1, increment (now H_Byte = 2)
N_3   ADDWF     0x43,W      ;WREG = 28 + 97 = BFH and C = 0
      DAW                   ;WREG = 25 and C = 1
      BNC       N_4         ;
      INCF      H_Byte,F    ;(now H_Byte = 3)
N_4   MOVWF     L_Byte      ;Now L_Byte = 25H
```

在代码执行后,文件寄存器的地址6=(03),WREG=25,这是因为71+88+69+97=325H。还可以用寄存器间接寻址方式和循环语句来更加高效地实现这个程序,这将在第6章中介绍。

5.1.8 无符号数的减法

许多微控制器都有两个不同的减法指令:SUB和SUBB。PIC18有4条减法指令:SUB-

LW、SUBWF、SUBWFB 和 SUBFWB。最后的两条是带借位的减法指令。注意,C 标志位被用作借位。下面将逐一介绍这 4 条减法指令。

1. SUBLW K 指令(WREG=K−WREG)

在减法运算中,PIC 微控制器(实际上是所有的 CPU)采用二进制取补的方法。既然每个 CPU 都有加法电路,那么设计独立的的减法电路就显得多余。因此,PIC 的加法电路也用来执行减法操作指令。假设 PIC 正在执行一条简单的减法指令,并且在该指令执行之前 C=0,于是可以将 CPU 对无符号数执行 SUBLW 指令的步骤归纳如下。

(1) 首先对减数做二进制补码运算(WREG 操作数)。

(2) 然后执行同被减数的加法(K 操作数)。

对于每一条 SUB 指令,无论操作数和寻址方式如何,CPU 内部硬件都必须执行这两步。然后,得到运算的结果,标志位被修改。例 5-5 说明了这两个步骤。

例 5-5 请给出下面程序代码的执行步骤。

```
MOVLW 0x23        ;load 23H into WREG (WREG = 23H)
SUBLW 0x3F        ;WREG = 3F - WREG
```

解:

```
  K    = 3F   0011 1111        0011 1111
- WREG = 23   0010 0011      + 1101 1101    (二进制补码)
         1C                  1 0001 1100
                        C = 1, D7 = N = 0   (结果为正数)
```

标志寄存器将被设置为:C=1,N=0(注意,D7 是负数的标志位)。程序员必须检测 N 或 C 标志位才能判断结果是正数还是负数。

在执行 SUB 指令之后,如果 N=0 或者 C=1,结果为正数;如果 N=1 或者 C=0,结果则为负数,最终获得的结果将是二进制的补码。一般情况下,结果都将以二进制补码的形式储存,但是也可以使用 NEGF 指令来改变它。在 PIC 中,另一条 SUB 指令是 SUBWF(目标值=fileReg−WREG),这将在例 5-6 中看到。

161

例 5-6 编制程序,执行 4C−6E。

解:

```
MYREG EQU 0x20
      MOVLW 0x4C        ;load WREG (WREG = 4CH)
      MOVWF MYREG       ;MYREG = 4CH
      MOVLW 0x6E        ;WREG = 6EH
      SUBWF MYREG,W     ;WREG = MYREG - WREG. 4C - 6E = DE, N = 1
      BNN   NEXT        ;if N = 0 (C = 1), jump to NEXT target
      NEGF  WREG        ;take 2's complement of WREG
NEXT  MOVWF MYREG       ;save the result in MYREG
```

下面是 SUBWF 指令执行后的具体步骤。

```
 4C   0100 1100                     0100 1100
-6E   0110 1110  二进制补码运算  =  1001 0010
-22                                 1101 1110
```

在执行 SUBWF 指令后，得到 N=1(或者 C=0)，结果是以二进制补码形式给出的负数。继续执行 NEGF 指令，实现二进制补码运算，最后得到结果为 MYREG=22H。

2. SUBWFB 带借位的减法(目的操作数＝WREG－W－借位)

SUBWFB 指令用于多字节数的减法运算，它考虑了来自低字节的进位。如果在执行 SUBWFB 指令前有 C=0，那么结果将减去 1。请参阅例 5-7。

3. SUBFWB 带借位的减法(目的操作数＝WREG－fileReg－$\overline{借位}$)

SUBFWB 指令也用于多字节数的减法运算，它也考虑了来自低字节的进位。请注意 SUBWFB 指令同 SUBFWB 指令之间的区别。附录 A 给出了这两条指令的具体描述。

例 5-7 编制程序，实现两个十六位数的减法运算。操作数是 2762H－1396H。假设文件寄存器地址 6=(62)，地址 7=(27)。请将所得的差存放在地址 6 和地址 7，其中地址 6 为低字节。

解：

```
loc 6 = (62)
loc 7 = (27)

MOVLW 0x96          ;load the low byte (WREG = 96H)
SUBWF 0x6,F   ;F = F - W = 62 - 96 = CCH, C = borrow = 0, N = 1
MOVLW 0x12          ;load the high byte (WREG = 12H)
SUBWFB 0x7,F        ;F = F - W - b̄, sub byte with the borrow
                    ;F = 27 - 12 - 1 = 14H
```

在执行 SUBWF 指令后，地址 6=62H－96H=CCH，进位标志位为 0，表明存在借位(因为N=1)。由于 C=0，在 SUBWFB 指令执行后，fileReg 地址 7＝27H－12H－1＝14H。于是，有 2762H－1296H＝14CCH。

162

5.1.9 PIC 减法的 C 标志位

注意，PIC18 和其他的 CPU(如 x86 和 8051)的不同之处在于对减法运算中的进位标志位的处理。其他的那些 CPU 自行对进位标志位做了处理，以至于通过检查 C 标志位就能确定结果的正负。在 PIC18 中，若 C=0，则结果为负，这就是为什么在带有借位的减法中会有 F=F－W－b。读者可以使用 MPLAB 仿真器来进一步理解这个问题。

5.1.10 无符号数的乘法

PIC 只支持单字节的乘法。假定单字节数为无符号数，则其语法如下。

```
MULLW   K     ;W × K and 16-bit is result is in PRODH:PRODL
```

在单字节乘法中，其中一个操作数必须放在 WREG 中，而另外一个操作数必须是立即数。在执行完乘法操作后，结果将被保存在 SFR 的 PRODH 和 PRODL 里，高字节放在 PRODH 中，而低字节放在 PRODL 中，如表 5-1 所示。下面是实现 25H×65H 的程序代码：

```
MOVLW 0x25          ;load 25H to WREG (WREG = 25H)
MULLW 0x65          ;25H * 65H = E99 where
                    ;PRODH = 0EH and PRODL = 99H
```

表 5-1 无符号数乘法小结(MULLW K)

乘 法	第一字节数	第二字节数	结 果
字节×字节	WREG	K	PRODH=高字节,PRODL=低字节

注意:多于 8 位数的乘法需要做适当的处理。

5.1.11 无符号数的除法

在 PIC18 中没有单独的除法指令。除法运算可以通过编制重复的减法程序来实现。在字节的除法运算中,被除数(分子)放在文件寄存器里,反复地减去除数(分母),减法的次数就是所得的商,而最后的余数则被保存在文件寄存器里。请看下面的例子。

```
NUM    EQU    0x19        ;set aside fileReg
MYQ    EQU    0x20
MYNMB  EQU    D'95'
MYDEN  EQU    D'10'
       CLRF   MYQ         ;quotient = 0
       MOVLW  MYNMB       ;WREG = 95
       MOVWF  NUM         ;numerator = 95
       MOVLW  MYDEN       ;WREG = denominator = 10
B1     INCF   MYQ,F       ;increment quotient for every 10 subtr
       SUBWF  NUM,F       ;subtract 10 (F = F - W)
       BC     B1          ;keep doing it until C = 0
       DECF   MYQ,F       ;once too many
       ADDWF  NUM,F       ;add 10 back to get remainder
```

163

5.1.12 除法的应用

有时候,ADC(模数转换器)被连接到单片机的一个端口,用来表示诸如温度或者压力之类的大小。8 位 ADC 在 00h~FFh 范围内给出的是十六进制数据。因此,必须将十六进制数转化成十进制数。接下来,例 5-8 采用重复地除以 10 并保存余数的方法来执行转换。

例 5-8 假设文件寄存器地址 0x15 的内容为 FD(十六进制)。编制程序,将其转换为十进制数,并将结果保存在地址 0x22、0x23 和 0x24,其中 0x22 存放最低位。

解:

```
#include <P18F458.INC>
;PIC Assembly Language Program for division (by repeated subtraction)
;(Byte/Byte)

NUME        EQU    0x15       ;RAM location for NUME
QU          EQU    0x20       ;RAM location for quotient
RMND_L      EQU    0x22
RMND_M      EQU    0x23
RMND_H      EQU    0x24
MYNUM       EQU    0xFD       ;FDH = 253 in decimal
MYDEN       EQU    D'10'      ;253/10
            ORG    0H         ;start at address 0
            MOVLW  MYNUM      ;WREG = 253, the numerator
            MOVWF  NUME       ;load numerator
            MOVLW  MYDEN      ;WREG = 10, the denominator
            CLRF   QU,F       ;clear quotient
```

```
D_1         INCF   QU,F          ;increment quotient for every sub
            SUBWF  NUME          ;sub WREG from NUME value
            BC     D_1           ;if positive go back (C = 1 for positive)
            ADDWF  NUME          ;once too many, this is our first digit
            DECF   QU,F          ;once too many for quotient
            MOVFF  NUME,RMND_L   ;save the first digit
            MOVFF  QU,NUME       ;repeat the process one more time
            CLRF   QU            ;clear QU
D_2         INCF   QU,F
            SUBWF  NUME          ;sub WREG from NUME value
            BC     D_2           ;(C = 1 for positive)
            ADDWF  NUME          ;once too many
            DECF   QU,F
            MOVFF  NUME,RMND_M   ;2nd digit
            MOVFF  QU,RMND_H     ;3rd digit
HERE        GOTO   HERE          ;stay here forever
            END                  ;end of asm source file
```

若要将一个十进制数转换为ASCII数，可使用OR指令，将十进制数同30H做或运算，请参阅6.4节和6.5节。

164

例5-9 分析例5-8中的程序代码，被除数为253。

解：

为了将二进制数(十六进制)转换为十进制数，采用重复除以10的方法直到商小于10。每次除法运算后，保存余数。对于8位二进制数，如FDH，其对应的十进制数为253，分析如下。

	商	系数	
253/10 =	25	3	（低位数字）
25/10 =	2	5	（中间数字）
		2	（高位数字）

因此，得到转换结果：FDH=253。为了能显示数据，需要将数据转换为ASCII数，这将在以后的章节中讨论。

5.1.13 复习题

1. 在PIC18中，对于两个字节数相乘的乘法指令，应该将一个字节数放在寄存器________中，而另外一个字节数使用立即数K。
2. 在无符号的单字节数乘法中，所得的积存放在寄存器________中。
3. MULLW F是合法的指令吗？请给出理由。
4. 在PIC18中，乘法指令可处理的最大两个数分别是________和________。
5. 判断对错：DAW指令只对WREG起作用。
6. DAW fileReg, d是合法的指令吗？请给出理由。
7. 指令ADDLW K将所得的和存放在________。
8. 为什么下面的指令是非法的：ADDLW fileReg?
9. 重新编写上面的指令，将WREG加到fileReg中。
10. 指令ADDWFC fileReg, W将所得的和存放在________中。
11. 确定下面指令执行后，标志位DC和C的状态。

(a) MOVLW 0x4F　　(b) MOVLW 0x9C
　　ADDLW 0xB1　　　　ADDLW 0x63

12. 请说明 CPU 执行减法 05H－43H 的过程。
13. 如果在指令 SUBFWB fileReg, F 执行之前有 C=1，WREG=95H，fileReg=4FH，那么在执行减法运算以后，WREG 和 fileReg 中的内容分别是多少？

5.2 有符号数的概念及其算术运算

目前讨论的数据都是无符号数，这意味着整个 8 个二进制位都是用来表示数据大小的，而许多应用中都需要用有符号数。本节将讨论有符号数的概念以及相关的一些指令。如果你的应用不涉及有符号数，可以跳过本节的学习。

5.2.1 计算机中有符号数的概念

在日常生活中，所用到的数字可以是正数，也可以是负数。例如，零下 5℃的温度可以表示成－5℃，零上 20℃的温度则可以表示成＋20℃。这要求计算机必须能够处理这些数。为此，计算机科学家设计了以下的方法来表示正数和负数。最高有效位(MSB)被单独地用来表示数据的符号(正或负)，其余的位则被用来表示数据的大小。符号位为 0 表示正号，而符号位为 1 则表示负号。下面将讨论有符号的字节表示。

5.2.2 有符号的 8 位操作数

在有符号的字节操作数中，D7(MSB)位是符号位，而 D0～D6 用来表示数字的大小。若 D7＝0，则操作数是正的；若 D7 ＝1，则操作数为负的。状态寄存器的标志位 N 为 D7 位。

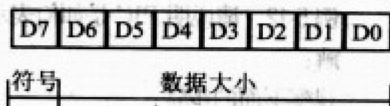


图 5-2 8 位有符号操作数

5.2.3 正数

如图 5-2 所示格式的正数的取值范围是 0～＋127。如果正数大于＋127，那么就需要用到 16 位操作数。由于 PIC18 不支持 16 位的数据，所以在此不讨论这个问题。

```
  0     0000 0000
 +1     0000 0001
 ...       ...
 +5     0000 0101
 ...       ...
+127    0111 1111
```

5.2.4 负数

对于负数，D7 等于 1；然而数据的大小是用二进制补码表示的。虽然汇编器会完成这个转换，但是了解这种转换仍然很有必要。为了转换成负数的表示形式(二进制补码)，需要遵循以下步骤。

(1) 用 8 位二进制数来表示数的大小(无符号)。
(2) 将各二进制位按位取反。
(3) 对其执行加 1 操作。

上述3个步骤的应用，请参阅例5-10、例5-11和例5-12。

例5-10 请说明PIC是如何表示−5的。

解：

观察下面的步骤：

(1) 0000 0101 以8位二进制数表示的数字5

(2) 1111 1010 按位取反

(3) 1111 1011 加1，结果为FB(十六进制)

因此，−5=FBH，即−5对应的二进制由符号数补码表示。D7=N=1表示该数是负数。

例5-11 请说明PIC是如何表示−34H的。

解：

观察下面的步骤：

(1) 0011 0100 以二进制表示的34H

(2) 1100 1011 按位取反

(3) 1100 1100 加1，结果为CC(十六进制)

因此，−34=CCH，即−34H对应的二进制由符号数补码表示。D7=N=1表示该数是负数。

例5-12 请说明PIC是如何表示−128的。

解：

观察下面的步骤：

(1) 1000 0000 以8位二进制数表示的数字5

(2) 0111 1111 按位取反

(3) 1000 0000 加1，结果为80(十六进制)

因此，−128=80H，即−128对应的二进制有符号数补码表示。D7=N=1表示该数是负数。注意，128(二进制数为10000000)的无符号表示与−128(二进制数为10000000)的无符号表示是相同的。

167

从以上的例子中可以清楚地发现，单字节负数的取值范围是−1～−128。下面列出了单字节有符号数的取值范围。

十进制	二进制	十六进制
-128	1000 0000	80
-127	1000 0001	81
-126	1000 0010	82
...	....	..
-2	1111 1110	FE
-1	1111 1111	FF
0	0000 0000	00
+1	0000 0001	01
+2	0000 0010	02
..	....	..
+127	0111 1111	7F

以上的数据范围解释了在第3章讨论过的BNZ和其他条件分支指令中相对地址为−128～+127的奥秘。

5.2.5 有符号数运算中的溢出问题

在使用有符号数的时候，必须要处理的一个严重问题就是溢出问题。PIC 能用 OV (溢出)标志位来表征这个错误，但是程序员仍必须随时留意结果是否出错，因为 CPU 只能识别 0 和 1，而没有人类的正负数概念。那到底什么是溢出呢？如果有符号数运算的结果超出了寄存器的范围，就会产生溢出，这一点是程序员必须要注意的。下面来看例 5-13。

例 5-13 阅读下面的程序代码，分析运行结果，包括标志为 N 和 OV。

```
MOVLW +D'96'        ;WREG = 0110 0000
ADDLW +D'70'        ;WREG = (+96) + (+70) = 1010 0110
                    ;WREG = A6H = -90 decimal, INVALID!!
```

解：

```
     +96          0110 0000
+    +70          0100 0110
+    166          1010 0110      N = 1 (负数) 且 OV = 1。所得的和是 - 90。
```

CPU 运算结果为负(N=1)，这是错误的。CPU 将 OV 置 1 表示出现溢出错误。记住：N 标识位为 D7 位；如果 N=0，其结果为正；如果 N=1，其结果为负。

在例 5-13 中，根据 CPU 的结果，+96 与+70 相加的和竟是−90。为什么呢？究其原因，在于两数相加的结果超出了 WREG 所能容纳的范围。像所有的 8 位寄存器一样，WREG 的正数上限是+127。CPU 的设计者们专门设置了溢出标志位，以此来提醒程序员运算结果的出错。标志位 N 就是结果的 D7 位。若 N=0，则和是正数。若N=1，则和就是负数。

5.2.6 何时设置 OV 标志位

在 8 位有符号数的运算中，当以下两个条件中的任何一个条件成立时，OV 将被置 1：

(1) 从 D6 向 D7 有进位，而 D7 是没有进位的(C=0)；

(2) D7 有进位(C=1)，而从 D6 到 D7 是没有进位的。

换而言之，如果从 D6 到 D7 有进位或者 D7 有进位，但两者并不是同时发生，那么溢出标志位将被置 1。这就意味着，如果从 D6 到 D7 有进位，同时 D7 也有进位，那么 OV=0。在例 5-13中，因为从 D6 到 D7 有进位，但 D7 是没有进位的，所以 OV=1。研究例 5-14、例 5-15 和例 5-16，可以深入理解有符号算术运算中的溢出标志位。

例 5-14 观察下面的程序代码，注意标志为 OV 和 N 的作用。

```
MOVLW -D'128'       ;WREG = 1000 0000 (WREG = 80H)
ADDLW -D'2'         ;W = (-128) + (-2)
                    ;W = 1000000 + 11111110 = 0111 1110,
                    ;N = 0, W = 7EH = +126, invalid
```

解：

```
   -128      1000 0000
+   -2       1111 1110
   -130      0111 1110    N=0（正数），OV=1
```

根据 CPU 的运算，结果为+126，这是错误的，标志位 OV=1 说明了这一点。

例 5-15　观察下面的程序代码，注意标志为 OV 和 N 的作用。

```
MOVLW -D'2'        ;WREG = 1111 1110 (WREG = FEH)
ADDLW -D'5'        ;WREG = (-2) + (-5) = -7 or F9H
                   ;correct, since OV = 0
```

解：

```
    -2       1111 1110
+   -5       1111 1011
    -7       1111 1001   OV=0,N=1。所得的和是正确的。
```

169

根据 CPU 的运算，结果为−7，这是正确的结果，标志位 OV=0 说明了这一点。

例 5-16

```
MOVLW +D'7'        ;WREG = 0000 0111
ADDLW +D'18'       ;W = (+7) + (+18)
                   ;W = 00000111 + 00010010 = 0001 1001
                   ;W = (+7) + (+18) = +25, N = 0, positive and
                   ;correct, OV = 0
```

解：

```
    +7       0000 0111
+   +18      0001 0010
    +25      0001 1001   N=0（正数 25）且 OV=0
```

根据 CPU 的运算，结果为+25，这是正确的结果，标志位 OV=0 正好说明了这一点。

从以上例子中可以得出结论：在任何有符号数的加法中，标志位 OV 表征结果是否有效。若 OV=1，则结果就是错误的；若 OV=0，则结果是有效的。需要重点指出的是，在无符号数的加法中，程序员必须监视 C（进位标志位）的状态，而在有符号数的加法中，程序员则必须监视 OV（溢出标志位）的状态。在 PIC 中，一些指令（如 BNC、BC）允许程序在执行无符号数加法后进行分支转移，如 5.1 节所示。另外，BOV 和 BNOV 指令用来修正有符号数中的错误。此外，对于 N 标志位，还有两条可用的分支指令：BN 和 BNN。

5.2.7　二进制补码运算指令

PIC18 有一个特殊的指令来对某个数进行二进制补码运算，它就是指令 NEG fileReg。下一节将讨论这个问题。

5.2.8 复习题

1. 在 8 位操作数中，位________用作符号位。
2. 将－16H 转换为它的二进制补码。
3. 字节长的有符号数的范围是________。
4. 给出＋9 和－9 的二进制表示。
5. 请解释进位和溢出的区别。

5.3 逻辑和比较指令

除了 I/O 指令和算术指令以外，逻辑指令是使用最广泛的指令之一。在本节中，将讨论布尔逻辑指令，如 AND、OR、XOR 和取补指令，此外还将学习比较指令。

5.3.1 AND 指令

```
ANDLW K   ;WREG = WREG AND K
```

上面的这条指令将对两个操作数进行逻辑与运算，结果放在 WREG 寄存器里。指令 ANDWF fileReg,d 的目的操作数可以是 fileReg 或 WREG。fileReg 操作数可以是 RAM 文件寄存器中的任何寄存器，详情请参阅附录 A。AND 指令会影响标志位 Z 和 N。标志位 N 是结果的 D7 位，若结果为 0，则 Z=1。AND 指令经常用来对操作数的指定位做掩码运算(及置 0 操作)。请看下面的例 5-17。

逻辑与功能

输入		输出
X	Y	X 逻辑与 Y
0	0	0
0	1	0
1	0	0
1	1	1



例 5-17 请给出下面指令代码的运行结果。

```
MOVLW  0x35        ;WREG = 35H
ANDLW  0x0F        ;W = W AND 0FH (now W = 05)
```

解：

```
35H  0 0 1 1 0 1 0 1
0FH  0 0 0 0 1 1 1 1
05H  0 0 0 0 0 1 0 1   ;35H AND 0FH = 05H, Z = 0, N = 0
```

5.3.2 OR 指令

```
IORLW K      ;WREG = WREG Inclusive-OR K
```

上面的这条指令将对两个操作数进行逻辑或运算，结果放在 WREG 寄存器里。指令 IORWF fileReg,d 的目的操作数可以是 fileReg 或 WREG。fileReg 操作数可以是 RAM 文件寄存器里的任何寄存器。详情请参阅附录 A。OR 指令会影响标志位 Z 和 N。标志位 N 是结果的 D7

逻辑或功能

输入		输出
X	Y	X 逻辑或 Y
0	0	0
0	1	1
1	0	1
1	1	1



位，若结果为0，则Z=1。OR指令经常用来对操作数的指定位做置1操作。请171看例5-18。

例5-18 (a) 请给出下面的程序代码的结果：

```
MOVLW 0x04                    ;WREG = 04
IORLW 0x30                    ;now WREG = 34H
```

(b)假设端口*B*的RB2位用来控制门外的灯，RB5位用来控制屋内的灯。请编制程序，打开门外的灯和关闭屋内的灯。

解：

```
(a) 04H     0000 0100
    30H     0011 0000
    34H     0011 0100        04 OR 30 = 34H, Z = 0 and N = 0

(b) BCF    TRISB,2           ;make RB2 an output
    BCF    TRISB,5           ;make RB5 an output
    MOVLW  B'00000100'       ;D2 = 1
    IORWF  PORTB,F           ;make RB2 = 1 only
    MOVLW  B'11011111'       ;D5 = 0
    ANDWF  PORTB,F           ;mask RB5 = 0 only
```

5.3.3 EX-OR 指令

```
XORLW K      ;WREG = WREG XOR K
```

上面的这条指令将对两个操作数进行逻辑EX—OR(异或)运算，结果放在WREG寄存器里。另一条异或运算指令XORWF fileReg, d的目的操作数可以是fileReg或WREG。fileReg操作数可以是RAM文件寄存器里的任何寄存器。详情请参阅附录A。OR指令会影响标志位Z和N。标志位N是结果的D7位，若结果为0，则Z=1。请看例5-19和例5-20。

逻辑异或功能

输入		输出
A	*B*	*A*逻辑异或*B*
0	0	0
0	1	1
1	0	1
1	1	0



EX—OR指令也可以用来检查两个寄存器的值是否相同。指令XORWF fileReg,W将对WREG寄存器的内容和文件寄存器地址的内容进行异或操作，结果存在WREG里。如果它们的值相等，那么WREG内容为0。之后，可以根据这个结果，用BZ指令172来做出决策。请看例5-20和例5-21。

EX—OR指令的另外一个广泛的应用就是对操作数的各二进制位取反，如：

```
MOVLW 0xFF          ;WREG = FFH
XORWF PORTC,F       ;EX-OR PORTC with 1111 1111 will
                    ;change all the bits of Port C to
                    ;opposite
```

例5-19 请给出下面指令代码的运行结果。

```
MOVLW 0x54
XORLW 0x78
```

解：

```
54H        0 1 0 1 0 1 0 0
78H        0 1 1 1 1 0 0 0
2CH        0 0 1 0 1 1 0 0    54H XOR 78H = 2CH, Z = 0, N = 0
```

例 5-20 通过与已知数的异或运算，EX－OR 指令可用来测试寄存器的内容。在下面的程序代码中，对 45H 进行自我逻辑异或运算，将会产生标志位 Z。

```
OVER  MOVF   PORTB,W      ;get a byte from PORTB into WREG
      XORLW  0x45
      BNZ    OVER         ;branch if not zero
```

解：

```
45H        01000101
45H        01000101
00         00000000
```

将一个数与自身进行逻辑异或运算，结果为零且 Z=1。可以使用 BNZ 指令来判断。若与其他数进行逻辑异或运算，则结果将是一个非零数。

例 5-21 读取 PORTB，并测试其值是否为 45H。若是，则向 PORTC 发送数据 99H，否则就退出。

解：

```
      CLRF   TRISC        ;Port C = output
      CLRF   PORTC        ;Port C = 00
      SETF   TRISB        ;Port B = input
      MOVLW  0x45
      XORWF  PORTB,W      ;EX-OR with 0x45, Z = 1 if yes
      BNZ    EXIT         ;branch if PORTB has value other than 0
      MOVLW  0x99
      MOVWF  PORTC        ;Port C = 99h
EXIT:...
```


5.3.4 COMF 指令（将 fileReg 取反）

COMF 指令用来对文件寄存器的内容取反。取反操作是将 0 变成 1，将 1 变成 0，这就是所谓的一元取反。

逻辑取反功能

输入	输出
X	NOTX
0	1
1	0

X —▷o— NOT X

```
CLRF   TRISB        ;Port B = Output
MOVLW  0x55
MOVWF  PORTB
COMF   PORTB,F      ;now PORTB = AAH
```

5.3.5 NEGF 指令（将 fileReg 取补 ）

NEGF 指令用来对文件寄存器的内容进行二进制补码转换，如例 5-22 所示。

例 5-22 计算 85H 的二进制补码数。注意,85H 就是十进制数−123。

解:

```
MYREG EQU 0x10
      MOVLW 0x85                  85H = 1000 0101
      MOVWF MYREG                 1'S = 0111 1010
      NEGF  MYREG                           + 1
                      2's comp        0111 1011 = 7BH
```

5.3.6 比较指令

PIC18 有 3 条用于比较运算的指令,如表 5-2 所示。这些比较运算指令将比较文件寄存器和 WREG 中的值,然后根据它们的大小关系来执行相应的操作。比较指令事实上是一条减法运算指令,只不过在比较过程中操作数的值并不会改变。在 PIC18 中,比较指令的执行不会影响状态寄存器的标志位。必须再次强调的是,不管比较的结果如何,比较指令都不会影响操作数。下面用相应的例子来讨论表 5-2 中的每一条指令。

表 5-2 PIC18 比较指令

CPFSGT	比较 FileReg 和 WREG,若 FileReg 大于 WREG,则跳过下一条指令
CPFSEQ	比较 FileReg 和 WREG,若 FileReg 等于 WREG,则跳过下一条指令
CPFSLT	比较 FileReg 和 WREG,若 FileReg 小于 WREG,则跳过下一条指令

注意:这些比较指令不会影响状态寄存器的标志位,也不会改变 fileReg 和 WREG 的值。

5.3.7 CPFSGT 指令

CPFSGT 指令用来比较文件寄存器 WREG 中的值。如果文件寄存器的值大于 WREG 的值,就跳过下一条指令。请看图 5-3 和例 5-23。

174

例 5-23 编制程序,比较数值 27 和 54 的大小,并将其中较大的数放在文件寄存器的地址 0x20。

解:

```
VAL_1 EQU   D'27'
VAL_2 EQU   D'54'
GREG  EQU   0x20
MOVLW  VAL_1        ;WREG = 27
MOVWF  GREG         ;GREG = 27
MOVLW  VAL_2        ;WREG = 54
CPFSGT GREG         ;skip if GREG > WREG
MOVWF  GREG         ;place the greater in GREG
```

5.3.8 CPFSEQ 指令

CPFSEQ 指令用来比较文件寄存器和 WREG 中的值,如果文件寄存器的值等于 WREG 的值,就跳过下一条指令。请看图 5-4 和例 5-24。

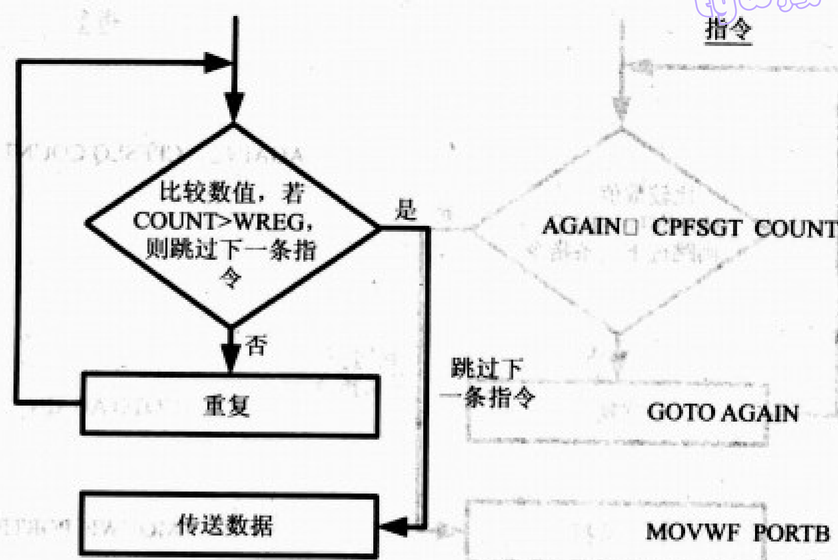


图 5-3 CPFSGT 指令的流程图

例 5-24 编制程序，不断地检测 PORTD 是否为 63H。当且仅当 PORTD=63H 时，程序停止。

解：

```
        SETF    TRISD       ;PORTD = input
        MOVLW   0x63        ;WREG = 63H
BACK    CPFSEQ  PORTD       ;skip BRA instruction if PORTD = 63H
        BRA     BACK
```


5.3.9 CPFSLT 指令

CPFSLT 指令用来比较文件寄存器和 WREG 中的值，如果文件寄存器的值小于 WREG 的值，就跳过下一条指令。请看图 5-5 和例 5-25。

例 5-25 编制程序，比较数值 27 和 54 的大小，并将其中较小的数放在文件寄存器的地址 0x20。

解：

```
VAL_1 EQU   D'27'
VAL_2 EQU   D'54'
LREG  EQU   0x20   ;location for smaller of two

MOVLW  VAL_1       ;WREG = 27
MOVWF  LREG        ;LREG = 27
MOVLW  VAL_2       ;WREG = 54
CPFSLT LREG        ;skip if LREG < WREG
MOVWF  LREG        ;place the smaller value in LREG
```

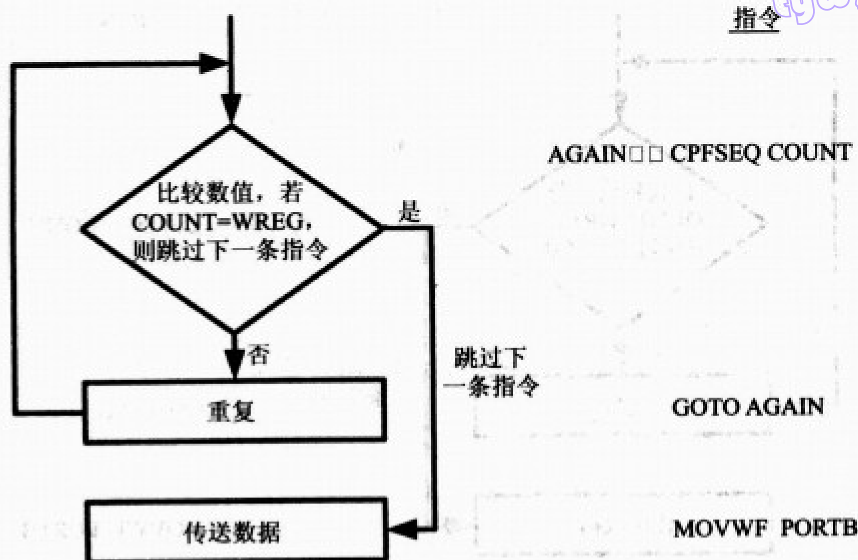


图 5-4　CPFSEQ 指令的流程图

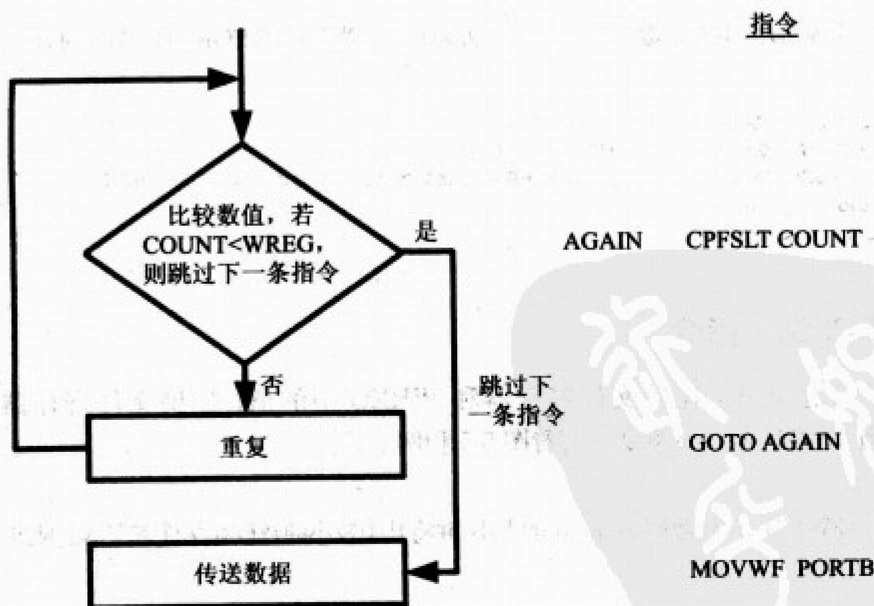


图 5-5　CPFSLT 指令的流程图

例 5-26　假设端口 D 为输入端口，且连接到一个温度传感器。编制程序，读取温度数值，并检测它是否为 75。根据检测结果，按照下面的要求将测得的温度值保存到寄存器中。

如果 $T=75$，那么 WREG=75；

如果 $T>75$，那么 GREG=T；

如果 $T<75$，那么 LREG=T。

解：

```
LREG EQU 0x20
GREG EQU 0x21
        SETF     TRISD              ;PORTD = input
        MOVLW    D'75'              ;WREG = 75 decimal
        CPFSGT   PORTD              ;skip BRA instruction if PORTD > 75
        BRA      LEQ
        MOVFF    PORTD, GREG
        BRA      OVER
LEQ     CPFSLT   PORTD              ;skip if PORTD < 75
        BRA      OVER
        MOVFF    PORTD, LREG
OVER    .....                       ;it must be equal, WREG = 75
```


例5-27 编制程序，判断 PORTB 中的数据是否为 99H。若是，则向 PORC 写字母 Y；否则向 PORTC 写字母 N。

解：

```
        CLRF     TRISC              ;PORTC = output
        MOVLW    A'N'               ;WREG = 'N' (ASCII)
        MOVWF    PORTC              ;PORTC = 'N'
        SETF     TRISB              ;PORTB = input
        MOVLW    0x99               ;WREG = 99H
        CPFSEQ   PORTB              ;skip BRA instruction if PORTB = WREG
        BRA      OVER
        MOVLW    'Y'
        MOVWF    PORTC              ;PORTC = 'Y'
OVER    .....
```

5.3.10 复习题

1. 执行下面的指令，确定寄存器的内容。
   (a) MOVLW 0x37
       ANDLW 0xCA
   (b) MOVLW 0x37
       IORLW 0xCA
   (c) MOVLW 0x37
       XORLW 0xCA
2. 要实现对 WREG 中某些位的掩码(置零)，则需要将其同已知数________进行逻辑与运算。
3. 要实现对 WREG 中某些位的置 1，则需要将其同已知数________进行逻辑或运算。
4. 将一个数同自身进行逻辑异或运算，则结果是________。
5. 判断对错：CPFSLT 指令会改变操作数的值。
6. 在下面的程序代码中，要跳过 BRA 指令的执行，则 MYREG 中的内容应为多少呢？

```
        MOVLW   0x99
BACK    CPFSLT  MYREG
        BRA     BACK
```

7. 执行下面的代码，确定 WREG 寄存器的内容。

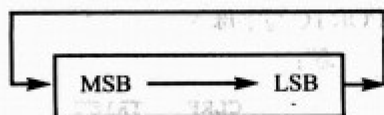
```
MOVLW  0
IORLW  0x99
XORLW  0xFF
```

5.4 移位指令和数据串行化

在许多应用中，都需要用到操作数的移位操作。在 PIC18 中，移位操作指令 RRCF、RRNCF、RLCF 和 RLNCF 就是针对这个目的而设计的。它们可用来对文件寄存器中的数据进行左移位或右移位操作。移位指令的应用十分广泛，接下来讨论它们的使用。移位操作有两种类型。一种是文件寄存器的简单移位，而另一种是带有进位的移位。下面将分别阐述它们的使用。

5.4.1 文件寄存器的左移或右移操作

```
RRNCF fileReg,d ;rotate fileReg right(no carry)
```



在右移操作中，文件寄存器的 8 位都将被右移一位，最低有效位 D0 被移到最高有效位 D7 位。在执行移位操作后，结果保存在 fileReg 或者 WREG 中，究竟保存在哪里将取决于 d 位。请看下面的程序代码。

```
MREG EQU 0x20
      MOVLW 0x36              ;WREG = 0011 0110
      MOVWF MYREG
      RRNCF MYREG,F           ;MYREG = 0001 1011
      RRNCF MYREG,F           ;MYREG = 1000 1101
      RRNCF MYREG,F           ;MYREG = 1100 0110
      RRNCF MYREG,F           ;MYREG = 0110 0011
```

```
      RLNCF fileReg,d   ;rotate fileReg left (no carry)
```



在左移操作中，文件寄存器的 8 位将被左移一位，最高有效位 D7 被移到最低有效位 D0 位。在执行移位操作后，结果保存在 fileReg 或者 WREG 中，究竟保存在哪里将取决于 d 位。请看下面的程序代码。

```
MREG EQU 0x20
      MOVLW 0x72              ;WREG = 0111 0010
      MOVWF MYREG

      RLNCF MYREG,F           ;MYREG = 1110 0100
      RLNCF MYREG,F           ;MYREG = 1100 1001
```

179

注意，指令 RRNCF 和 RLNCF 都会影响到标志位 Z 和 N。

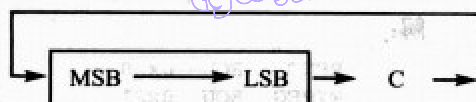
5.4.2 带进位的移位

在 PIC18 中，还有两条涉及进位标志位的移位指令。下面将分别介绍这两条指令。

```
RRCF fileReg, d   ;rotate fileReg right through carry
```

在 RRCF 中，随着二进制的从左向右移动，进位标志位被移到 MSB(最高有效位)，同时 LSB(最低有效位)被移出到进位标志位。换而言之，在 RRCF 中，标志位 C 被移到 MSB 位，同

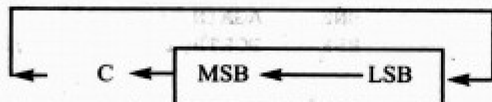
时 LSB 位被移出到标志位 C。实际上，进位标志位就好像是寄存器的组成部分，使该寄存器成为一个 9 位的寄存器。




```
MREG EQU 0x20
        BCF     STATUS,C     ;make C = 0 (carry is D0 of status)
        MOVLW   0x26         ;WREG = 0010 0110
        MOVWF   MYREG
        RRCF    MYREG,F      ;MYREG = 0001 0011 C = 0
        RRCF    MYREG,F      ;MYREG = 0000 1001 C = 1
        RRCF    MYREG,F      ;MYREG = 1000 0100 C = 1
```

```
        RLCF fileReg, d           ;rotate fileReg left through carry
```

在 RLCF 中，随着二进制的从右向左的移动，进位标志位被移到 LSB(最低有效位)，同时 MSB(最高有效位)被移出到进位标志位。换而言之，在 RLCF 中，标志位 C 被移到 LSB 位，同时 MSB 位被移出到标志位 C。请看下面的程序代码图示。进位标志位再次成为寄存器的一部分，使其成为一个 9 位的寄存器。




```
MREG EQU 0x20
        BSF     STATUS,C     ;make C = 1 (carry is D0 of status)
        MOVLW   0x15         ;WREG = 0001 0101
        MOVWF   MYREG
        RLCF    MYREG,F      ;MYREG = 0010 1011 C = 0
        RLCF    MYREG,F      ;MYREG = 0101 0110 C = 0
        RLCF    MYREG,F      ;MYREG = 1010 1100 C = 0
        RLCF    MYREG,F      ;MYREG = 0101 1000 C = 1
```

180

5.4.3 串行化数据

串行化数据是一种传送数据的方式，它通过微控制器的一个引脚每次发送一个二进制数据位。实现数据串行化的传输方法有以下两种。

(1) 使用串行端口。在用串行端口传送数据时，程序员对数据传送顺序的控制是相当有限的。串行端口数据传输将在第 10 章中讨论。

(2) 每次传送一个二进制位，并且控制数据的顺序和位置。在许多新一代的设备(如 LCD、ADC、ROM 等)中，串行传输方式相当流行，因为它在印制电路板上所占空间比较小。

下面将讨论如何使用移位操作指令来实现数据的串行化传输。

5.4.4 字节数据的串行化

串行化数据是移位指令的最广泛运用之一。使用移位指令，可以将数据进行串行化传输(每次传输一位)。例 5-28 说明了如何使用 PIC 的引脚来串行化传输整个字节的数据。

例 5-28 编制程序，通过 TB1 引脚串行化传输 41H，在数据的起始和末尾设置高电平。先传输 LSB 位。

解：

```
RCNT    EQU  0x20          ;fileReg loc for counter
MYREG   EQU  0x21          ;fileReg loc for rotate

        BCF    TRISB,1     ;make RB1 an output bit
        MOVLW  0x41        ;WREG = 41
        MOVWF  MYREG       ;value to be serialized
        BCF    STATUS,C    ;C = 0
        MOVLW  0x8         ;counter
        MOVWF  RCNT        ;load the counter
        BSF    PORTB,1     ;RB1 = high
AGAIN   RRCF   MYREG,F     ;rotate right via carry
        BNC    OVER
        BSF    PORTB,1     ;set the carry bit to PB1
        BRA    NEXT
OVER    BCF    PORTB,1
NEXT    DECF   RCNT,F
        BNZ    AGAIN
        BSF    PORTB,1     ;RB1 = high
```

181

例5-29演示了如何串行传输一个字节的数据(每次传输一位)。在第16章中将会学习如何把这些思想应用于串行RTC(实时时钟)芯片。例5-30将介绍如何扫描一个字节数据的二进制位。

例5-29　编制程序，将一个字节的数据在RC7引脚串行地输入(每次传输一位)，并将结果保存在文件寄存器的地址0x21。先传输字节数据的LSB位。

解：

```
RCNT  EQU  0x20          ;fileReg loc for counter
MYREG EQU  0x21          ;fileReg loc for incoming byte

      BSF    TRISC,7     ;make RC7 an input bit
      MOVLW  0x8         ;counter
      MOVWF  RCNT        ;load the counter
AGAIN BTFSC  PORTC,7     ;skip if RC7 = 0
      BSF    STATUS,C    ;carry = 1
      BTFSS  PORTC,7     ;skip if RC7 = 1
      BCF    STATUS,C    ;otherwise carry = 0
      RRCF   MYREG,F     ;rotate right carry into MYREG
      DECF   RCNT,F      ;decrement the counter
      BNZ    AGAIN       ;repeat until RCNT = 0
                         ;now loc 21H has the byte
```

例5-30　编制程序，统计给定字节数据中为1的二进制位的数量。

解：

```
R1     EQU  0x20  ;fileReg loc for number of 1s
COUNT  EQU  0x21  ;fileReg loc for counter
VALREG EQU  0x22  ;fileReg loc for the byte

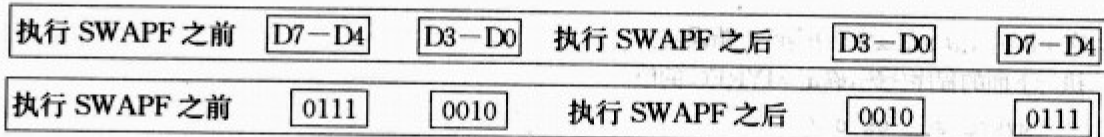
       BCF   STATUS,C   ;C = 0
       CLRF  R1         ;R1 keeps the number of 1s
       MOVLW 0x8        ;counter = 08 to rotate 8 times
       MOVWF COUNT
       MOVLW 0x97       ;find the number of 1s in 97H
       MOVWF VALREG
```

```
AGAIN RLCF   VALREG,F     ;rotate it through the C once
      BNC    NEXT         ;check for C
      INCF   R1,F         ;if C = 1 then add one to R1 reg
NEXT  DECF   COUNT,F
      BNZ AGAIN           ;go through this 8 times
                          ;now loc 0x20 has the number of 1s
```

182

5.4.5 SWAPF fileReg, d

PIC 还有另外一个有用的指令 SWAPF，它仅对文件寄存器进行操作，将低 4 位和高 4 位进行交换。换而言之，低 4 位被放到了高 4 位，高 4 位被放到了低 4 位。请看下面的示意图和例 5-31。



例 5-31 (a) 执行下面的代码，确定 MYREG 寄存器的内容。

(b) 如果不使用 SWAPF 指令，如何将字节数据的高 4 位和低 4 位交换呢？编制简单的程序，实现上面的要求。

解：

(a)

```
MYREG EQU 0x20
      MOVLW 0x72          ;WREG = 72H
      MOVWF MYREG         ;MYREG = 72H
      SWAPF MYREG,F       ;MYREG = 27H
```

(b)

```
MYREG EQU 0x20
      MOVLW 0x72          ;WREG = 0111 0010
      MOVWF MYREG         ;MYREG = 0111 0010
      RLNCF MYREG,F       ;MYREG = 1110 0100
      RLNCF MYREG,F       ;MYREG = 1100 1001
      RLNCF MYREG,F       ;MYREG = 1001 0011
      RLNCF MYREG,F       ;MYREG = 0010 0111
```

5.4.6 复习题

1. 执行下面的程序代码，确定文件寄存器中 MYREG 的内容。

```
MYREG EQU 0x40
MOVLW 0x25
MOVWF MYREG
RRNCF MYREG,F
RRNCF MYREG,F
RRNCF MYREG,F
RRNCF MYREG,F
```

183

2. 执行下面的程序代码，确定文件寄存器中 MYREG 的内容。

```
MYREG EQU 0x40
MOVLW  0x25
MOVWF  MYREG
RLNCF  MYREG,F
RLNCF  MYREG,F
RLNCF  MYREG,F
RLNCF  MYREG,F
```

3. 执行下面的程序代码,确定 MYREG 的值。

```
MYREG EQU 0x40
CLRF   MYREG
BSF    STATUS,C        ;C = 1
RRCF   MYREG,F
BSF    STATUS,C        ;C = 1
RRCF   MYREG,F
```

4. 在 PIC 中,指令 RLCF W 会产生错误吗?
5. 执行下面的程序代码,确定 MYREG 的值。

```
MYREG EQU 0x40
MOVLW  0x85
MOVWF  MYREG
SWAPF  MYREG,F
```

5.5 BCD 和 ASCII 码转换

本节将通过实际的例子来说明如何使用算术和逻辑指令。在以后的章节中还将讨论它们在实际设备中的应用。比如,许多新型的微控制器都带有一个实时时钟(RTC),时间和数据都被保存在这里面,无论何时断电都没有影响。这些微控制器提供用 BCD 码表示的时间和数据。然而,为了显示这些 BCD 数,必须先将 BCD 数转换成 ASCII 码。接下来,将介绍逻辑指令和移位指令在转换 BCD 码和 ASCII 码中的应用。

5.5.1 ASCII 数

在 ASCII 键盘上,当键 0 被按下时,0110000(30H)将被送到计算机中。同理,当键 1 被按下时,31H(0110001)将被传送到计算机中,以此类推,如表 5-3 所示。

必须注意的是,虽然 ASCII 码是美国(还有其他的许多国家)的标准,但是 BCD 码的使用还是很普遍的。因为键盘、打印机、显示器都使用 ASCII 码,那么如何将数据从 ASCII 码转换成 BCD 码呢?这是下面要讨论的问题。

表 5-3 数字 0~9 的 ASCII 码和 BCD 码

键	ASCII(十六进制)	二进制	BCD(非压缩码)
0	30	011 0000	0000 0000
1	31	011 0001	0000 0001
2	32	011 0010	0000 0010
3	33	011 0011	0000 0011
4	34	011 0100	0000 0100
5	35	011 0101	0000 0101

（续）

键	ASCII（十六进制）	二进制	BCD（非压缩码）
6	36	011 0110	0000 0110
7	37	011 0111	0000 0111
8	38	011 1000	0000 1000
9	39	011 1001	0000 1001

5.5.2 从压缩 BCD 码到 ASCII 码的转换

在许多系统中都有一个被称作实时时钟(RTC)的单元。不管上电与否，RTC 都将连续地提供时间(时、分、秒)以及日期(年、月、日)。但是，这些数据是以压缩 BCD 码的形式表示的。为了将这些数据在设备(如 LCD)上显示，或者在打印机上打印出来，就必须将其转换成 ASCII 码的格式。

为了将压缩 BCD 码转换成 ASCII 码，必须首先将它转换成非压缩 BCD 码。于是，非压缩 BCD 码被标识为 011 0000(30H)。下面将说明如何将压缩 BCD 码转换成 ASCII 码，请看例 5-32。

```
压缩 BCD 码      非压缩 BCD 码        ASCII

29H             02H  & 09H         32H  & 39H
0010 1001       0000 0010 &        0011 0010 &
                0000 1001          0011 1001
```

例 5-32 假设 WREG 寄存器的内容是压缩 BCD 码。编制程序，将压缩 BCD 码转换成两个 ASCII 码，并将结果分别放置在寄存器地址 6 和地址 7。

解：

```
BCD_VAL EQU 0x29
L_ASC   EQU 0x06   ;set aside file register location
H_ASC   EQU 0x07   ;set aside file register location

      MOVLW BCD_VAL      ;WREG = 29H, packed BCD
      ANDLW 0x0F         ;mask the upper nibble (W = 09)
      IORLW 0x30         ;make it an ASCII, W = 39H ('9')
      MOVWF L_ASC        ;save it (L_ASC = 39H ASCII char)
      MOVLW BCD_VAL      ;W = 29H get BCD data once more
      ANDLW 0xF0         ;mask the lower nibble (W = 20H)
      SWAPF WREG,W       ;swap nibbles (WREG = 02H)
      IORLW 0x30         ;make it an ASCII, W = 32H ('2')
      MOVWF H_ASC        ;save it (H_ASC = 32H ASCII char)
```


5.5.3 从 ASCII 码到压缩 BCD 码的转换

为将 ASCII 码转换成压缩 BCD 码，首先必须将它转换成非压缩 BCD 码，然后将它组合成压缩 BCD 码。比如，在键盘上按下的 4 和 7，将分别产生 34H 和 37H。转换的目标是产生压缩 BCD 码：47H 或者 0100 0111。下面将说明实现这种转换的过程。

键	ASCII(十六进制)	非压缩 BCD 码	压缩 BCD 码
4	34	00000100	
7	37	00000111	01000111 which is 47H

```
MYBCD EQU 0x20    ;set aside location in file register

      MOVLW A'4'        ;WREG = 34H, hex for ASCII char 4
      ANDLW 0x0F        ;mask upper nibble (WREG = 04)
      MOVWF MYBCD       ;save it in MYBCD loc
      SWAPF MYBCD,F     ;MYBCD = 40H
      MOVLW A'7'        ;WREG = 37H, hex for ASCII char 7
      ANDLW 0x0F        ;mask upper nibble (WREG = 07)
      IORWF MYBCD,F     ;MYBCD = 47H, a packed BCD
```

在这个转换结束后，ASCII 码就转换成了压缩 BCD 码，结果将会是压缩 BCD 码的形式。正如本章前面所看到的，DAW 这条特殊的指令需要处理的数据形式就是压缩 BCD 码。

5.5.4 复习题

1. 对于下面的十进制数，分别给出它们的压缩 BCD 码和非压缩 BCD 码。
   (a) 15　　(b) 99
2. 请分别给出数值 76 的二进制表示和十六进制表示，以及 BCD 码表示。
3. 在下面的指令执行后，WREG 寄存器中的内容是 BCD 码形式的吗？
   ```
   MOVLW D'54'
   ```
4. BCD 数 67H 在转换成 ASCII 码后分别是________ H 和________ H。
5. 下面的程序可以将 WREG 寄存器中的内容转换成 ASCII 码吗？
   ```
   MOVLW 0x09
   ADDLW 0x30
   ```

小结

本章讨论了 PIC 中有符号数和无符号数的算术运算指令。无符号数用全部的 8 位二进制来表示，其取值范围是 0～255(十进制)。有符号数使用 7 位来表示数的大小，使用最高位来表示数的符号，其取值范围是－128～＋127(十进制)。

BCD 码(二进制编码的十进制数)可用来表述数字 0～9。本章讨论了两种 BCD 码表示法：压缩 BCD 码和非压缩 BCD 码。PIC 单片机提供关于 BCD 码算术运算的特殊指令。

186

在编写 PIC 的算术运算程序时，要特别注意进位和溢出的条件。

本章还定义了关于逻辑 AND、OR、XOR 和取补运算，并介绍了实现这些逻辑运算的指令。同时，也介绍了比较指令和条件跳过指令。在位操作中，这些指令经常被用到。

PIC 的移位指令和半字节交换指令在许多应用场合(如串行设备)都有使用。本章还讨论了 BCD 码和 ASCII 码，以及它们之间的相互转换。

习题

1. 试确定下面程序代码的标志位 C、Z 和 DC 的状态。

```
(a) MOVLW    0x3F
    ADDLW    0x45
(b) MOVLW    0x99
    ADDLW    0x58
(c) MOVLW    0xFF
    MOVWF    MYREG
    BSF      STATUS,C
    MOVLW    0
    ADDWFC   MYREG,F
(d) MOVLW    0xFF
    ADDLW    0x1
(e) MOVLW    0xFE
    MOVWF    MYREG
    BSF      STATUS,C
    MOVLW    0
    ADDWFC   MYREG,F
(f) BCF      STATUS,C
    MOVLW    0xFF
    MOVWF    MYREG
    MOVLW    0
    ADDWFC   MYREG,F
```

2. 编制程序,将你的身份证的所有数字相加,并将结果保存在文件寄存器中。要求结果用BCD码表示。
3. 编制程序,将下面的几个数相加,并将结果保存在文件寄存器中:
   0x25、0x59 和 0x65
4. 将第3题的结果用BCD码表示。
5. 编制程序,(a)向文件寄存器的地址20H~23H写入数值25H;(b)将这些RAM地址的数据相加,并将结果保存在RAM地址60H中。
6. 请陈述实现下面运算的减法指令的步骤。
   (a) 23H−12H　(b) 43H−53H　(c) 99−99
7. 对于第6题,编制程序,实现每一个运算操作。
8. 判断对错:指令 DAW 只对 WREG 寄存器工作。
9. 编制程序,将7F9AH与BC48H相加,并将结果保存在起始地址为40H的RAM地址中。
10. 编制程序,将BC48H减去7F9AH,并将结果保存在起始地址为40H的RAM地址中。
11. 编制程序,将BCD数7795H与BCD数9548H相加,并将结果保存在起始地址为40H的RAM地址中。
12. 请说明如何在PIC18中实现77×34的运算。
13. 请说明如何在PIC18中实现77除以3的运算。
14. 判断对错:MULLW指令可以在PIC18的所有寄存器中运行。
15. MULLW指令将运算结果保存在寄存器________和________中。
16. 请给出下面数值的编译器表示。
    (a) −23　(b) +12　(c) −28　(d) +6FH　(e) −128　(f) +127
17. 在计算机中存储器的地址是________(有符号,无符号)数。

18. 编制程序，实现下面的算术运算，并指出标志位 OV 的状态。
    (a) (+15)+(−12)　　(b) (−123)+(−127)
    (c) (+25H)+(+34H)　　(d) (−127)+(+127)
19. 请解释标志位 C 和 OV 的区别，并指出它们分别用在哪里。
20. 标志位 OV 在什么情况下会被设置？请给出理由。
21. 哪一个寄存器含有 OV 标志位？
22. 在 PIC18 中如何检测 OV 标志位呢？又是如何检测 C 标志位的呢？
23. 假设 WREG=F0H。执行下面的程序。请指出运算结果以及所存放的寄存器。
    备注：下面的运算是相互独立的。

```
(a) ANDLW  0x45        (b) IORLW  0x90
(c) XORLW  0x76        (d) ANDLW  0x90
(e) XOWLW  0x90        (f) IORLW  0x90
(g) ANDLW  0xFF        (h) IORLW  0x99
(i) XORLW  0xEE        (j) XORLW  0xAA
```

24. 执行下面的指令，确定寄存器 WREG 中的内容。

```
(a) MOVLW   0x65
    ANDLW   0x76
(b) MOVLW   0x70
    IORL    0x6B
(c) MOVLW   0x95
    XORLW   0xAA
(d) MOVLW   0x5D
    ANDLW   0x78
(e) MOVLW   0x0C5
    IORLW   0x12
(f) MOVLW   0x6A
    XORLW   0x6E
(g) MOVLW   0x37
    IORLW   0x26
```

25. 判断对错：在 CPFSEQ 指令中，必须使用 WREG 作为其中的一个寄存器吗？
26. 请解释 CPFSGT 指令是如何工作的。
27. 比较指令会影响到状态寄存器的标志位吗？

28. 假设 MYREG=85H。在下面的程序中，执行比较指令后，程序是否会跳过下一条指令？

```
(a) MOVLW   0x90
    CPFSGT  MYREG
    INCF    MYREG,F
    ADDLW   0x2
(b) MOVLW   0x70
    CPFSGT  MYREG
    INCF    MYREG,F
    ADDLW   0x2
```

```
(c) MOVLW   0x85
    CPFSEQ  MYREG
    INCF    MYREG,F
    ADDLW   0x2
(d) MOVLW   0x5D
    CPFSLT  MYREG
    INCF    MYREG, F
    ADDLW   0x2
```

29. 在第 28 题中,请指出 MYREG 中的内容。
30. 执行下面的指令,确定寄存器 WREG 中的内容。

```
(a) MOVLW   0x56
    MOVWF   MYREG
    SWAPF   MYREG, F
    RRCF    MYREG, F
    RRCF    MYREG, F
(b) MOVLW   0x39
    BCF     STATUS,C
    MOVWF   MYREG, F
    RLCF    MYREG, F
    RLCF    MYREG, F
(c) BCF     STATUS,C
    MOVLW   0x4D
    MOVWF   MYREG
    SWAPF   MYREG, F
    RRCF    MYREG, F
    RRCF    MYREG, F
    RRCF    MYREG, F
(d) BCF     STATUS,C
    MOVLW   0x7A
    MOVWF   MYREG
    SWAPF   MYREG, F
    RLCF    MYREG, F
    RLCF    MYREG, F
```

31. 请使用下面的指令,编写代替 SWAPF 指令的程序。
    (a) 使用右移指令。
    (b) 使用左移指令。
32. 编制程序,确定 8 位数据中 0 的个数。
33. 编制程序,确定 8 位数据中第一个 1 的位置。扫描顺序是从 D0 到 D7。所得的结果放在 68H。
34. 编制程序,确定 8 位数据中第一个 1 的位置。扫描顺序是从 D7 到 D0。所得的结果放在 68H。
35. 步进电机使用下面顺序排列的二进制数来驱动电机。如何产生这些顺序的数据呢?
    1100,0110,0011,1001

36. 编制程序,将下列的压缩BCD数转换成ASCII数。将ASCII数保存在起始地址为40H的RAM空间。

189

```
MYBCD_1      EQU    0x76
MYBCD_2      EQU    0x87
```

37. 编制程序,将下列的ASCII数转换成压缩BCD数。将所得的BCD数保存在起始地址为60H的RAM空间。

```
MYASC_1      EQU    A'8'
MYASC_2      EQU    A'7'
MYASC_3      EQU    A'9'
MYASC_4      EQU    A'2'
```

复习题答案

5.1 节

1. WREG　2. PRODH和PRODL　3. 否。应该是指令`MULWF fileReg,F`。
4. 255和255。　5. 正确。　6. 否。DAW仅仅对WREG寄存器工作。
7. WREG　8. 不能将立即数、WREG和fileReg三者混淆。
9. 指令`ADDWF fileReg,F`　10. WREG
11. (a) WREG=00,C=1,DC=1
    (b) WREG=FF,C=0,DC=0

12.
```
   43H  0100 0011                      0100 0011
 - 05H  0000 0101  二进制补码        + 1111 1011
   3EH                                 0011 1110
```

13. fileReg=95H−4FH−0=46H,WREG=95H

5.2 节

1. D7
2. 16H的二进制表示是00010110,它的补码是11101010或者−16H=EA(十六进制)。
3. −128～+127。　4. +9=00001001和−9=11110111或者E7(十六进制)。
5. 溢出是指符号位(D7)中的进位,而进位则是位于寄存器之外的进位标志位。

5.3 节

1. (a) 02　(b) FFH　(c) FDH　2. 零。　3. 1　4. 所有的零。　5. 错误。

190

6. 小于0x99的任意值。　7. 66H。

5.4 节

1. 52H　2. 52H　3. C0H　4. 否。因为WREG是SFR。　5. 58H

5.5 节

1. (a) 15H=0001 0101压缩BCD码,0000 0001,0000 0101非压缩BCD码
   (b) 99H=1001 1001压缩BCD码,0000 1001,0000 1001非压缩BCD码
2. 3736H=00110111 00110110B,用BCD码表示为76H=0111 0110B。
3. 否。需要把它写成54H或者01010100B,以变成BCD码。不带H的54将被编译器当作36H处理。

191

4. 36H,37H　5. 是的。因为WREG=39H。

第 6 章
存储区转换、表处理、宏和模块

学习目标：

❑ PIC18 微控制器的所有寻址方式
❑ 各种寻址方式的对比和比较
❑ 每种寻址方式的 PIC 汇编语言指令编写
❑ 运用各种寻址方式访问 RAM 中的文件寄存器
❑ PIC18 的查表程序编写
❑ ROM 空间内固定数据的访问
❑ 宏和模块的创建
❑ PIC18 全部 4 KB 内存空间的访问
❑ PIC18 全部 16 个存储区的地址分配
❑ PIC18 所有存储区的访问
❑ PIC18 存储区的切换
❑ ASCII 码和 BCD 码转换的 PIC18 程序编写
❑ 创建和测试校验和的 PIC18 程序编写
❑ 编程过程中使用宏和模块的优点

CPU 可以通过不同的方式访问数据。数据可以位于寄存器或者存储器里，也可以作为立即数。这些不同的访问数据的方式被称作*寻址方式*。本章将讨论 PIC18 的寻址方式，并给出适当的例子。

微控制器的各种寻址方式在设计时就已经确定，因此程序员是无法修改的。PIC18 提供 4 种不同的寻址方式，它们是：

(1) 立即寻址
(2) 直接寻址
(3) 寄存器间接寻址
(4) 变址寻址

6.1 节将介绍立即寻址和直接寻址方式。6.2 节将讨论如何使用寄存器间接寻址方式访问数据存储器。6.3 节将讨论如何访问程序 ROM 中的固定数据和查询表。6.4 节将讨论文件寄存器 RAM 空间的位寻址能力。第 6.5 节讨论了存储区的转换，以及怎样访问当前访问存储区以外的存储区。关于校验和的生成和 BCD-ASCII 转换将在 6.6 节中讨论。6.7 节将介绍宏和模块，并描述模块化编程。

6.1 立即寻址与直接寻址方式

在这一节中，首先介绍立即寻址方式，然后再介绍直接寻址方式。

6.1.1 立即寻址方式

在立即寻址方式中，操作数是常数。顾名思义，在指令汇编时，操作数紧跟在操作码之后。注意，立即数在PIC里被称作常数。该寻址方式可以用来向WREG寄存器和选择的寄存器加载数据，但并不是对所有文件寄存器。立即寻址方式也可用于算术和逻辑指令。请看下面的例子。

```
MOVLW 0x25          ;load 25H into WREG
SUBLW D'62'         ;subtract WREG from 62
ANDLW B'01000000'   ;AND WREG with 40H
```

可以使用伪指令EQU来访问立即数，如下面的例子。

```
COUNT EQU 0x30
...    ..
MOVLW COUNT          ;WREG = 30h
```

注意，还可以使用立即寻址方式来执行算术和逻辑操作，但仅限于WREG寄存器。例如，指令ADDLW 0x25将25H加到WREG寄存器中。

194

6.1.2 直接寻址方式

正如第2章所提到的，256 B的访问存储区文件寄存器被分成两部分：分配给通用寄存器的低地址(00～7FH)和用于SFR的高地址(F80～FFFH)。当PIC18上电后，访问存储区是默认的存储区体。它是最小的存储区，所有PIC18处理器都具备。指令MOVFF就是用来选择存储区的。在6.5节讨论存储区转换时会再讨论它。

使用直接寻址或寄存器间接寻址方式可以访问整个数据存储区的文件寄存器。寄存器间接寻址方式将在下一节讨论。在直接寻址方式中，操作数位于已知地址的RAM存储空间里，其中操作数的地址作为指令的一部分给出。与立即寻址方式(指令中的字母L代表常数)相比，直接寻址指令中的字母F则说明文件寄存器的地址。请看下面的例子，并注意指令中的F。

```
MOVLW 0x56          ;WREG = 56H (immediate addressing mode)
MOVWF 0x40          ;copy WREG into fileReg RAM location 40H
MOVFF 0x40,0x50     ;copy data from loc 40H to 50H.
```

后面两条指令采用的是直接寻址方式。如果仔细地分析操作码，将会发现操作数的地址已嵌入在指令中，如图6-1a所示。

如图6-1b所示，8位地址的寻址范围为00～FFH。用于存储区转换的A位将在6.4节讨论。当然，在程序中使用名字来代替地址要容易得多，这在前几章已经举了很多的例子。必须要注意的是，文件寄存器数据RAM不支持立即寻址方式。换而言之，要想把数据传送到任何文件寄存器，都必须先把数据传送到WREG寄存器，然后再使用指令MOVWF将数据从WREG寄存器传送到文件寄存器中。

Address	00	02	04	06	08	0A	0C	0E	ASCII
0000	0E56	6E40	C040	F050	D7FF	FFFF	FFFF	FFFF	V.@n@.P. ....
0010	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	....
0020	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	....
0030	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	....

(a) MOVFF 直接寻址操作码

0110	111A	ffff	ffff

0≤ffff ffff≤FF

A-用于操作的存储区选择
A=0，使用默认存储区
A=1，使用由BSR(存储区选择寄存器)指定的存储区

(b) MOVWF 直接寻址方式

图 6-1 直接寻址

6.1.3 指令INCF fileReg,W与INCF fileReg,F的区别

在直接寻址方式中，当对文件寄存器执行操作时，可以选择把操作结果存放在原来的文件寄存器中或者存放在WREG中。这种选择在PIC编程中常常会导致意外错误发生，因此必须强调其正确的用法。下面的程序代码使用直接寻址方式将地址20H的内容加1，但存放结果的目的寄存器由参数W或者F决定。

```
MOVLW  0          ;WREG = 0
MOVWF  0x20       ;loc 0x20 = (0), WREG = 0
INCF   0x20,W     ;loc 0x20 = (0), WREG = 1
INCF   0x20,W     ;loc 0x20 = (0), WREG = 1
INCF   0x20,W     ;loc 0x20 = (0), WREG = 1
INCF   0x20,F     ;loc 0x20 = (1), WREG = 1
INCF   0x20,F     ;loc 0x20 = (2), WREG = 1
INCF   0x20       ;loc 0x20 = (3), WREG = 1
INCF   0x20       ;loc 0x20 = (4), WREG = 1
INCF   0x20,W     ;loc 0x20 = (4), WREG = 5
```

注意，在上面的代码中，当没有给出第二个参数时，则默认为文件寄存器(F)。

6.1.4 DECFSZ指令和DECF指令

此处需要考查的另外两条指令是DECFSZ和DECF。其中的任意一条都可用于循环操作。在指令DECFSZ fileReg,d中，将文件寄存器的值减1，若结果为0，则跳过下一条指令。但是，DECF不会跳过下一条指令。对比下面的两段程序代码，它们的功能都是对端口B取反5次。使用指令DECFSZ来实现循环操作的程序代码如下。

```
      CLRF    TRISB      ;Port B as output
      MOVLW   5          ;WREG = 5
      MOVWF   MYREG      ;counter = 5
      CLRF    PORTB      ;clear Port B
B1    COMF    PORTB      ;complement Port B
      DECFSZ  MYREG,F    ;decrement and skip if MYREG = 0
      GOTO    B1         ;go back since it is not zero
      SETF    PORTB      ;make PB = FFH
```

而下面的代码是使用指令 BNZ(若不为 0 则分支) 来实现循环操作。

```
        CLRF    TRISB           ;Port B as output
        MOVLW   5               ;WREG = 5
        MOVWF   MYREG           ;counter = 5
        CLRF    PORTB           ;clear Port B
B2      COMF    PORTB           ;complement Port B
        DECF    MYREG,F         ;decrement counter
        BNZ     B2              ;go back if MYREG is not zero
        SETF    PORTB           ;make PB = FFH
```

196

注意,在 BNZ 程序中,如果用指令 DECF MYREG,W 代替指令 DECF MYREG, F,那么将会陷入死循环,因为 MYREG 和 WREG 的值将永远保持不变,分别是 MYREG = 5 和 WREG = 4。

6.1.5 SFR 及其地址

PIC18 的端口 A、端口 B 等寄存器是寄存器组的一部分,通常被称作 SFR。PIC 有许多 SFR,并被广泛地使用,这将在后面的章节中讨论。SFR 可以通过名字(更容易使用) 或地址来访问。例如,端口 B 的地址是 F81H,端口 C 的地址是 F82H,如表 6-1 所示。注意下面的各指令组是如何实现同样的功能的。

```
MOVWF  0xF81        ;is the same as
MOVWF  PORTB        ;which means copy WREG into Port B

CLRF  0xF82         ;is the same as
CLRF  PORTC         ;which means clear Port C

BSF  0xFD8,0        ;is the same as
BSF  STATUS,C       ;which make C = 1
```

表 6-1 部分 PIC18SFR 的地址

符号	名称	地址
WREG	工作寄存器	FE8H
PORTA	端口 A	F80H
PORTB	端口 B	F81H
PORTC	端口 C	F82H
LATA	端口 A 的输出锁存器	F89H
LATB	端口 B 的输出锁存器	F8AH
LATC	端口 C 的输出锁存器	F8BH
TRISA	端口 A 的数据传送方向	F92H
TRISB	端口 B 的数据传送方向	F93H
TRISC	端口 C 的数据传送方向	F94H
INDF0	间接寻址寄存器 0	FEFH
INDF1	间接寻址寄存器 1	FE7H
FSR0L	间接数据存储器地址指针 0 的低 8 位	FE9H
FSR0H	间接数据存储器地址指针 0 的高 8 位	FEAH
FSR1L	间接数据存储器地址指针 1 的低 8 位	FE1H
FSR1H	间接数据存储器地址指针 1 的高 8 位	FE2H
PLUSW0	间接变址寄存器 0	FEBH

(续)

符 号	名 称	地 址
PREINC0	前增量寄存器 0	FECH
POSTDEC0	后减量寄存器 0	FEDH
POSTINC0	后增量寄存器 0	FEEH
TBLPTRL	表指针的低 8 位	FF6H
TBLPTRH	表指针的高 8 位	FF7H
TBLPTRU	表指针的更高字节位	FF8H
TABLAT	程序存储器表锁存器	FF5H
STATUS	状态标志寄存器	FD8H

表 6-1 列出了部分 PIC18SFR 及其对应的地址，但要注意下面的两点。

(1)SFR 的地址范围为 F80H～FFFH。这些地址在 FFFH 以下，因为 PIC18 从 FFFH 开始分配 SFR 地址，然后地址值逐渐减小，直到芯片所支持的 SFR 都被分配完。并不是 PIC18 系列的所有芯片都有同样的外设，因此，用于 SFR 的地址数也各不相同。

(2)并不是 F80H～FFFH 的所有地址空间都用于 SFR。未使用的地址作为保留地址，程序员不能使用。

例 6-1 编制程序，向端口 *B* 发送数据 55H。可以使用：

(a) 寄存器的名字；

(b) 它们的地址。

解：

(a)
```
CLRF   TRISB        ;Port B output
MOVLW  0x55         ;WREG = 55H
MOVWF  PORTB        ;Port B = 55H
```

(b) 查表 6-1 可知，TRISB 的地址为 F93H，PORTB 的地址为 F81H。

```
CLRF   0xF93        ;Port B output
MOVLW  0x55H        ;WREG = 55H
MOVWF  0xF81        ;Port B = 55H
```

关于 PIC18 的直接寻址方式，要注意下面的几点。

(1) 如果研究汇编程序的列表文件.1st，将发现 SFR 的名字都被表 6-1 所列的地址取代。

(2)WREG 寄存器是 SFR 之一，地址为 FE8H。

(3) 直接寻址方式也被称作寄存器直接寻址，这种命名是相对于下一节要讨论的寄存器间接寻址方式而言。

6.1.6 复习题

1. 微控制器的程序员可以设计新的寻址方式吗?
2. 请给出指令，将二进制数 1000 0000 传送到 WREG 寄存器中。
3. 为什么指令 MOVLF myvalue,fileReg 是非法的?

4. 判断对错：在 PIC18 中，PC(program counter，程序计数器）是 SFR。

198

5. 判断对错：在 PIC18 中，WREG 寄存器不是 SFR。

6.2　寄存器间接寻址方式

寄存器直接寻址或间接寻址方式可以用来访问文件寄存器的通用 RAM 空间。上一节已经介绍了直接寻址方式（又称作寄存器直接寻址）的使用。寄存器间接寻址方式在 PIC18 中是一种很重要的寻址方式。在这一节中将全面地讨论间接寻址方式。

6.2.1　寄存器间接寻址方式

在寄存器间接寻址方式中，寄存器被用作指针，指向数据 RAM 地址。PIC18 有 3 个这种寄存器，即 FSR0、FSR1 和 FSR2。FSR 是文件选择寄存器（file Select Register）的缩写；千万不要把 FSR 同 SFR 混为一谈。FSR 是 12 位的寄存器，可以访问 PIC18 数据 RAM 的整个 4096 B 空间。指令 LFSR（装载 FSR）可用来装载 RAM 地址。也就是说，当 FSRx 作为指针使用时，它们必须首先装载 RAM 地址，格式如下。

```
LFSR 0, 0x30        ;load FSR0 with 0x30
LFSR 1, 0x40        ;load FSR1 with 0x40
LFSR 2, 0x6F        ;load FSR2 with 0x6F
```

因为 FSR0、FSR1 和 FSR2 都是 12 位的寄存器，所以只有把它们分成 8 位数据，才能适应 SFR 的地址空间。PIC18 就是这样实现的。FSR 分成低字节和高字节两部分，分别表示为 FSRxL 和 FSRxH，如表 6-1 所示。在表 6-1 中可以看到，FSR0L 和 FSR0H 分别代表 12 位寄存器 FSR0 的低字节和高字节。注意，FSRxH 只使用了低四位。与寄存器间接寻址方式有关的另一个寄存器是 INDF（间接寄存器）。FSR0、FSR1 和 FSR2 均有相应的 INDF 寄存器，分别是 INDF0、INDF1 和 INDF2。当要将数据传送到 INDFx 时，也就是要传送数据到 FSR 所指向的 RAM 地址。同样地，当要读取 INDF 寄存器的内容时，也就是读取 FSR 所指向的 RAM 地址的内容。请看下面的程序代码。

```
LFSR   0, 0x30      ;FSR0 = 30H RAM location pointer
MOVWF INDF0         ;copy contents of WREG into RAM
                    ;location whose address is held by
                    ;12-bit FSR0 register
```

6.2.2　寄存器间接寻址方式的优点

寄存器间接寻址方式的一个优点是，它可以动态地访问数据，而直接寻址方式是静态的。例6-2 给出了 3 种方法来实现将数据 55H 传送到 RAM 地址 40H ～ 50H。注意，在解法(b) 有两条指令被重复执行了多次。也可以使用这两条指令来创建一个循环程序，如解法(c) 所示。解法

199

(c) 是最高效的，这可能是使用了寄存器间接寻址方式的缘故。在例 6-2 中，必须使用指令 INCF FSR0L,F 来递增指针的值，因为 PIC18 没有 INCF FSR0,F 这样的指令。在直接寻址方式里不可能实现循环，这就是它与寄存器间接寻址的最大区别。例如，要传送位于数据 RAM 连续地址中的一组数据的话，使用寄存器间接寻址方式要比直接寻址方式更加高效、更加动态。如例 6-3 所示。

例 6-2 编制程序:将数据 55H 复制到 40H～50H 的 RAM 存储空间,要求使用:

(a) 直接寻址方式;

(b) 寄存器间接寻址方式(不使用循环);

(c) 使用循环。

解:

(a)

```
MOVLW   0x55        ;load WREG with value 55H
MOVWF   0x40        ;copy WREG to RAM location 40H
MOVWF   0x41        ;copy WREG to RAM location 41H
MOVWF   0x42        ;copy WREG to RAM location 42H
MOVWF   0x43        ;copy WREG to RAM location 43H
MOVWF   0x44        ;copy WREG to RAM location 44H
```

(b)

```
MOVLW   55H         ;load with value 55H
LFSR    0,0x40      ;load the pointer. FSR0 = 40H
MOVWF   INDF0       ;copy W to RAM loc FSR0 points to
INCF    FSR0L,F     ;increment pointer. Now FSR0 = 41H
MOVWF   INDF0       ;copy W to RAM loc FSR0 points to
INCF    FSR0L,F     ;increment pointer. Now FSR0 = 42H
MOVWF   INDF0       ;copy W to RAM loc FSR0 points to
INCF    FSR0L,F     ;increment pointer. Now FSR0 = 43H
MOVWF   INDF0       ;copy W to RAM loc FSR0 points to
INCF    FSR0L,F     ;increment pointer. Now FSR0 = 44H
MOVWF   INDF0       ;copy W to RAM loc FSR0 points to
```

(c)

```
     COUNT   EQU 0x10    ;location 10H for counter
     MOVLW   0x5         ;WREG = 5
     MOVWF   COUNT       ;load the counter, Count = 5
     LFSR    0,0x40      ;load pointer. FSR0 = 40H, RAM address
     MOVLW   0x55        ;WREG = 55h value to be copied
B1   MOVWF   INDF0       ;copy WREG to RAM loc SFR0 points to
     INCF    FSR0L,F     ;increment FSR0 pointer
     DECF    COUNT,F     ;decrement the counter
     BNZ     B1          ;loop until counter = zero
```

在上述程序运行后,使用 MPLAB 仿真器检查 RAM 地址的内容:

40 = (55)

41 = (55)

42 = (55)

43 = (55)

44 = (55)

例 6-3 假定 30H～34H 的 RAM 空间里有一串 ASCII 数据,如下所示。编制程序,读取每一个字符并传送到端口 B,每次读取并传送一个字节。编程时请使用:

(a) 直接寻址方式;

(b) 寄存器间接寻址方式。

30 =（'H'）

31 =（'E'）

32 =（'L'）

33 =（'L'）

34 =（'O'）

解：

(a) 使用直接寻址方式。

```
CLRF  TRISB             ;make Port B an output
MOVFF 0x30, PORTB       ;copy contents of loc 0x30 to PB
MOVFF 0x31, PORTB
MOVFF 0x32, PORTB
MOVFF 0x33, PORTB
MOVFF 0x34, PORTB
```

(b) 使用寄存器间接寻址方式。

```
      COUNTREG EQU 0x20       ;fileReg loc for counter
      CNTVAL EQU 5            ;counter value
      CLRF  TRISB             ;make Port B an output (TRSI = 0 = out)
      MOVLW CNTVAL            ;WREG = 5
      MOVWF COUNTREG          ;load the counter, Count = 5
      LFSR  2,0x30            ;load pointer. FSR2 = 30H, RAM address
B3    MOVF  INDF2,W           ;copy RAM loc FSR2 points at to WREG
      MOVWF PORTB             ;copy WREG to PORTB
      INCF  FSR2L             ;increment FSR2 to point at next loc
      DECF  COUNTREG,F        ;decrement counter
      BNZ   B3                ;loop until counter = zero
```

当使用MPLAB仿真器仿真上述程序时，请确保RAM地址30H～34H中的内容为HELLO。注意，指令MOVF INDF2,W将数据从INDF2传送到WREG寄存器中。

当使用MPLAB仿真本章的例子时，读者可能会注意到，无法查看INDF0、POSTDEC0或PLUSW0的内容。因为这些寄存器不是物理上可实现的RAM地址。访问这些寄存器需要使用间接寻址方式。关于物理上不可实现的寄存器，请参阅图2-4。 201

6.2.3 FSR的自动增量

FSR是一个12位的寄存器，取值范围为000H～FFFH，总共覆盖PIC18的全部4KB RAM空间。当地址（如5FFH）自增量时，使用指令INCF FSR0L,F来实现指针加1将会出现问题。指令INCF FSR0L,F不能将来自低位的进位传送给FSR1H寄存器。PIC18提供了用于FSRn自增量/自减量的方法。用于实现这些方法的指令CLRF的指令格式如表6-2所示。

表6-2 用于PIC18 FSRn自增量/自减量的CLRF指令

指令	功能描述
CLRF INDFn	清空FSRn所指的文件寄存器后，FSRn保持不变
CLRF POSTINCn	清空FSRn所指的文件寄存器后，FSRn增1
CLRF PREINCn	FSRn先增1，然后清空FSRn所指的文件寄存器
CLRF POSTDECn	清空FSRn所指的文件寄存器后，FSRn减1

（续）

指令	功能描述
CLRF PLUSW*n*	清空(FSR*n*＋WREG)所指寄存器，FSR*n* 和 W 的内容都保持不变

注意，表 6-2 给出了 CLRF 的指令格式，该格式同样也适用于其他所有类似的指令。自增量 / 自减量将影响整个 FSR*n* 的 12 位，但不会影响标志寄存器。这意味着即使 FSR0 从 FFF 递减到 000，标志寄存器也不会有任何反应。PLUSW*n* 在基于 RAM 的查表程序中用得很多，请参阅 6.4 节。

例 6-4 编制程序，清空从 60H 开始的 16 个 RAM 地址的内容。要求使用：

(a) 指令 INCF FSR*n*L；

(b) 自增量方法。

解：

(a)

```
COUNTREG EQU 0x10        ;fileReg loc for counter
CNTVAL EQU D'16'         ;counter value
        MOVLW   CNTVAL       ;WREG = 16
        MOVWF   COUNTREG     ;load the counter, Count = 16
        LFSR    1,0x60       ;load pointer. FSR1 = 40H, RAM address
B2      CLRF    INDF1        ;clear RAM loc FSR1 points to
        INCF    FSR1L,F      ;increment FSR1L, point to next loc
        DECF    COUNTREG,F   ;decrement counter
        BNZ     B2           ;loop until counter = zero
```

(b)

```
COUNTREG EQU 0x10        ;fileReg loc for counter
CNTVAL EQU D'16'         ;counter value
        MOVLW   CNTVAL       ;WREG = 16
        MOVWF   COUNTREG     ;load the counter, Count = 16
        LFSR    1,0x60       ;load pointer. SFR0 = 40H, RAM address
B3      CLRF    POSTINC1     ;clear RAM, increment FSR1 pointer
        DECF    COUNTREG,F   ;decrement counter
        BNZ     B3           ;loop until counter = zero
```

202

例 6-5 从 30H 开始的 RAM 地址中存放着一个 5B 大小的数据块。编制程序，将其复制到从 60H 开始的 RAM 地址单元中。

解：

```
        COUNTREG EQU 0x10  ;fileReg loc for counter
        CNTVAL EQU D'5'    ;counter value
        MOVLW   CNTVAL     ;WREG = 10
        MOVWF   COUNTREG   ;load the counter, count = 10
        LFSR    0,0x30     ;load pointer. FSR0 = 30H, RAM address
        LFSR    1,0x60     ;load pointer. FSR1 = 60H, RAM address
B3      MOVF    POSTINC0,W ;copy RAM to WREG and increment FSR0
        MOVWF   POSTINC1   ;copy WREG to RAM and increment FSR1
        DECF    COUNTREG,F ;decrement counter
        BNZ     B3         ;loop until counter = zero
```

在运行上述程序之前，有：

```
30 = ('H')  31 = ('E')  32 = ('L')  33 = ('L')  34 = ('O')
```

在运行上述程序后，地址 60H～64H 中的数据同地址 30H～40H 的数据相同：

```
30 = ('H')  31 = ('E')  32 = ('L')  33 = ('L')  34 = ('O')
60 = ('H')  61 = ('E')  62 = ('L')  63 = ('L')  64 = ('O')
```

例 6-6　假定 RAM 地址 40H～43H 有如下的十六进制数据。编制程序，对这些数据求和，并把结果存放到地址 0x06 和 0x07。

40 = (7D)，41 = (EB)，40 = (C5)，43 = (5B)

解：

```
        COUNTREG EQU 0x20       ;fileReg loc for counter
        L_BYTE EQU 0x06         ;fileReg loc for L_Byte
        H_BYTE EQU 0x07         ;fileReg loc for L_Byte
        CNTVAL EQU 4            ;counter value
        MOVLW   CNTVAL          ;WREG = 4
        MOVWF   COUNTREG        ;load the counter
        LFSR    0,0x40          ;load pointer. FSR0 = 40H, RAM address
        CLRF    WREG            ;clear WREG
        CLRF    H_BYTE          ;clear H_BYTE
B5      ADDWF   POSTINC0, W     ;add RAM to WREG and increment FSR0
        BNC     OVER            ;if C = 0, go to next
        INCF    H_BYTE,F        ;C = 1, add 1 to high byte
OVER DECF       COUNTREG,F      ;decrement counter
        BNZ     B5              ;loop until counter = zero
        MOVWF   L_BYTE
```

203

上面例子是第 5 章中的例 5-2 的寄存器间接寻址实现。请比较两者的不同。

要知道如何使用全部的 3 个 FSR*n* 寄存器，请研究和仿真例 6-7。

例 6-7　编制程序，将下面两个多字节的 BCD 数相加，并保存结果。

　　12896577H
＋　23647839H

解：

```
        COUNTREG EQU 0x20 ;fileReg loc for counter
        CNTVAL EQU D'4'   ;counter value
        MOVLW CNTVAL      ;WREG = 4
        MOVWF COUNTREG    ;load the counter. Count = 4
        LFSR  0,0x30      ;load pointer. FSR0 = 30H, RAM address
        LFSR  1,0x50      ;load pointer. FSR1 = 50H, RAM address
        LFSR  2,0x60      ;load pointer. FSR2 = 60H, RAM address
        BCF   STATUS,C    ;clear carry flag for the LSB
B3      MOVF  POSTINC0,W  ;copy RAM to WREG and INC FSR0
        ADDWFC POSTINC1,W ;add RAM to WREG and INC FSR1
        DAW               ;decimal adjust WREG
        MOVWF POSTINC2    ;copy WREG to RAM and INC FSR2
        DECF  COUNTREG,F  ;decrement counter
        BNZ   B3          ;loop until counter = zero
```

在执行加法运算之前，有：

```
高字节地址                                     低字节地址
33 = (12)   32 = (89)   31 = (65)   30 = (77)
53 = (23)   52 = (64)   51 = (78)   50 = (39)
```

在执行加法运算之后，有：

```
63 = (36)   62 = (54)   61 = (44)   60 = (16)
```

注意，上面使用了这样的惯例，即将低字节存放在低字节地址，把高字节存放在高字节地址。请单步运行上面的程序，考察 FSRx 和存储器的内容，进一步掌握寄存器间接寻址方式。

6.2.4 复习题

1. 指令 MOVWF 0x40 使用的是________寻址方式，请给出理由。
2. 分配给寄存器 FSR0L 的地址是多少？
3. 分配给寄存器 FSR0H 的地址是多少？
4. FSRn 是________位的寄存器。
5. 如果数据位于数据 RAM 文件寄存器，那么哪些寄存器可以用于寄存器间接寻址方式？

204

6.3 查询表与表处理

到现在为止，大家已知道 PIC18 带有 2 MB 的代码（程序）空间和 4 KB 的数据 RAM 空间。程序员从来都不会使用数据 RAM 空间来存放程序代码，但是却会使用程序代码空间来存放定值数据。在这一节中，将讨论如何访问驻留在程序 ROM 空间里的定值数据。首先要讨论的是如何在使用 DB（定义字节）伪指令定义的程序 ROM 中存放定值数据。

6.3.1 DB 伪指令和程序 ROM 中的定值数据

DB（定义字节，Define Byte）伪指令常用来分配以字节为单位的 ROM 程序（代码）空间。换而言之，DB 伪指令用来定义 8 位的定值数据。在使用 DB 伪指令定义定值数据时，数字可以是十进制、二进制、十六进制，或者是 ASCII 码。DB 伪指令还可以用来定义 ASCII 码字符串。请看例 6-8。在例 6-8 中要注意的是，对于单个字符必须使用单引号（'），而对于字符串则必须使用双引号（"）。

例 6-8 假定下面的定值数据已被烧进 PIC 芯片的程序 ROM 里。请指出从 500H 开始的每个 ROM 地址的内容。关于 ASCII 字符对应的十六进制值，请参阅附录 F。

```
;MY DATA IN ROM
      ORG 500H              ;notice it must be an even address
DATA1 DB D'28'              ;DECIMAL(1C in hex)
DATA2 DB B'00110101'        ;BINARY (35 in hex)
DATA3 DB 0x39               ;HEX

      ORG 510H              ;notice it must be an even address
DATA4 DB 'Y'                ;single ASCII char
DATA5 DB '2','0','0','5';ASCII numbers

      ORG 518H              ;notice it must be an even address
DATA6 DB "Hello ALI"        ;ASCII string
      END
```

解：

```
DATA1           DATA2           DATA3
500 = (1C)      501 = (35)      502 = (39)

DATA4           DATA5
510 = (59)      511 = (32)      512 = (30)      513 = (30)      514 = (35)
       Y               2               0               0               5

DATA6
518 = (48)      519 = (65)      51A = (6C)      51B = (6C)      51C = (6F)

       H               e               l               l               o
51D = (20)     51E = (41)     51F = (4C)     520 = (49)
    SPACE              A              L              I
```

MPASM 还允许使用 DATA 代替 DB，来定义 255(0xFF) ～ 65535(0xFFFF) 之间的定值数据。

205

6.3.2　PIC18 的读表操作

例 6-8 说明了如何向程序 ROM 中存放定值数据。存在的唯一问题是，2 MB 的程序(代码)空间的寻址是在程序计数器寄存器的直接控制下进行的。这意味着，需要使用一个 SFR 来指向要在代码空间中读取的数据。正因为如此，这种寻址方式就被称作寄存器间接 ROM 寻址方式。该寻址方式广泛地用来访问 PIC18 程序 ROM 空间的数据单元，这就是通常所说的表处理。

在 PIC18 中，有一组指令专门用于表处理，可以进行读表和写表操作。鉴于读表操作的广泛使用，让我们首先来讨论读表操作。表 6-3 列出了 PIC18 的读表指令。在读取定值的字节数据时，需要用到一个指向数据的地址指针和一个存放数据的寄存器。TBLPTR 是一个 21 位的寄存器，常用来指向要读取的数据。21 位的寄存器 TBLPTR 可以覆盖 PIC18 的整个 2 MB 的程序存储空间。剩下的问题是没有专门指令用于装载 21 位的地址。

表 6-3　PIC18 的读表指令

指　令	功　能	描　述
TBLRD*	读表操作	读表后，TBLPTR 保持不变
TBLRD*＋	读表，并后增量	读表后，TBLPTR 加 1
TBLRD*－	读表，并后增量	读表后，TBLPTR 减 1
TBLRD＋*	先增量，后读表	TBLPTR 加 1 后，再读表

注意：字节数据从 TBLPTR 指向的程序存储空间读进 TABLATch 寄存器。

TBLPTR 寄存器由 3 个 8 位字节组成，分别是 TBLPTRL(低字节)、TBLPTRH(高字节) 和 TBLPTRU(更高字节)，它们都是 SFR。注意，TBLPTRU 的最后 2 位是未用的，且均被清零。

TABLAT 是用于表处理的另一个 SFR。TABLAT 寄存器用来暂存读取出来即将被送往 CPU 的数据。请参阅例 6-9。在下面的几个例子中，由于表单元素存储在 PIC18 的第一个 64KB 地址空间(0000～FFFFH) 中，所以只用到 TBLPTRL 和 TBLPTRH 两个寄存器。如果数据的存储地址为 10000H 或更高地址，就必须装载 TBLPTRU 寄存器。

6.3.3 TBLPTR 的自动增量

由于 TBLPTR 是 21 位的寄存器，其取值范围为 000000H～1FFFFFH，可寻址 PIC18 的整个 2MB 的 ROM 空间。当自增量地址（如 5FFH）时，使用指令 INCF TBLPTRL，F 来实现指针加 1 就会出现问题。指令 INCF TBLPTRL，F 不能将来自低位的进位传送给 TBLPTRH 寄存器。PIC18 提供了几种自增量 / 自减量的方法供选择，如 TBLRD*＋（读取表数据，再增量）、TBLRD*－（读取表数据，再减量）等，如表 6-3 所示。更具体的例子请看例 6-10、例 6-11 和例 6-12。

206

例 6-9 在下面的程序中，假定单词 USA 已被烧录到从 500H 开始的 ROM 地址中，且程序已烧录到从 0 开始的 ROM 空间。请分析该程序是如何工作的，并说明程序执行后 USA 的存放位置。

解：

```
        ORG    0000H                ;burn into ROM starting at 0
        CLRF   TRISB                ;make PB an output
        MOVLW  0x0                  ;WREG = 0 look-up table low-byte addr
        MOVWF  TBLPTRL              ;look-up table low-byte addr
        MOVLW  0x05                 ;WREG = 5 look-up table high-byte addr
        MOVWF  TBLPTRH              ;look-up table high-byte addr
        TBLRD*                      ;TABLAT = 'U' char pointed to by TABPTR
        MOVFF  TABLAT,PORTB         ;send it to Port B
        INCF   TBLPTRL,F            ;TBLPTRL = 01 pointing to next (501)
        TBLRD*                      ;TABLAT = 'S' char pointed to by TBLPTR
        MOVFF  TABLAT,PORTB         ;send it to Port B
        INCF   TBLPTRL,F            ;TBLPTRL = 02 pointing to next (502)
        TBLRD*                      ;TABLAT = 'A' char pointed to by TBLPTR
        MOVFF  TABLAT,PORTB         ;send it to Port B
HERE    GOTO HERE                   ;stay here forever

;data is burned into code(program) space starting at 500H
        ORG    500H
MYDATA  DB     "USA"
        END                         ;end of program
```

Program Memory

Address	00	02	04	06	08	0A	0C	0E	ASCII
04F0	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	....
0500	5355	0041	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	USA..... ....
0510	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	....
0520	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	....

Opcode Hex | Machine | Symbolic

Special Function Registers

Address	SFR Name	Hex	Decimal	Binary	Char
0F80	PORTA	00	0	00000000	.
0F81	PORTB	41	65	01000001	A
0F82	PORTC	00	0	00000000	.
0F83	PORTD	00	0	00000000	.
0F84	PORTE	00	0	00000000	.

执行上面的程序后，ROM 地址 500H～502H 的内容为：

207

```
500 = ('U') 501 = ('S') 502 = ('A')
```

最初，TBLPTR = 500H(TBLPTRH = 05,TBLPTRL = 0)。指令 TBLRD* 传送地址 500H 的内容到 TABLAT 中。寄存器 TABLAT 的内容是 55H，即字符 'U' 的 ASCII 码。它将被传送到端口 B。接下来，TABLAT 的内容自动加 1 变成 501H。于是，TBLRD 指令将会读取下一个 ROM 地址的内容，即字符 'S'。程序执行完成后，将发送 'U'、'S' 和 'A' 的 ASCII 码到端口 B，每次发送一个字符。本程序的循环实现请参阅下一个例子。

例 6-10 假定在起始地址为 250H 的程序 ROM 里存放着 USA。编制程序，将所有的字符发送到端口 B，每次发送一个字节。

解：

(a) 使用计数器：

```
RCOUNT EQU 0x20                 ;counter loc in fileReg
CNTVAL EQU 0x3                  ;counter value
       ORG    0000H             ;burn into ROM starting at 0
       MOVLW 0x50               ;WREG = 50, low-byte addr
       MOVWF TBLPTRL            ;look-up table low-byte addr
       MOVLW 0x02               ;WREG = 2, high-byte addr
       MOVWF TBLPTRH            ;look-up table high-byte addr
       MOVLW CNTVAL             ;WREG = 03, counter value
       MOVWF RCOUNT             ;load counter
       CLRF  TRISB              ;TRSIB = 00 (Port B as output)
B6     TBLRD*                   ;read table byte pointed to by TBLPTR
       MOVFF TABLAT,PORTB       ;send it to Port B
       INCF  TBLPTRL,F          ;increment to point to next char
       DECF  RCOUNT,F           ;dec the counter
       BNZ   B6                 ;repeat if counter not zero
HERE   GOTO  HERE               ;stay here
;data is burned into code(program) space starting at 250H
       ORG    0x250
MYDATA DB     "USA"
       END
```

(b) 使用空字符作为字符串结束的标志：

```
       ORG    0000H             ;burn into ROM starting at 0
       MOVLW 0x50               ;WREG = 50, low-byte addr
       MOVWF TBLPTRL            ;look-up table low-byte addr
       MOVLW 0x02               ;WREG = 2, high-byte addr
       MOVWF TBLPTRH            ;look-up table high-byte addr
       CLRF  TRISB              ;TRSIB = 00 (Port B as output)
B7     TBLRD*                   ;bring in next byte
       MOVF  TABLAT,W           ;copy to WREG (Z = 1, if null)
       BZ    EXIT               ;is it null char? exit if yes
       MOVWF PORTB              ;send it to Port B
       INCF  TBLPTRL,F          ;increment pointing to next
       BRA   B7                 ;continue
EXIT   GOTO  EXIT
       ORG   0x250
MYDATA DB    "USA",0            ;notice null
       END
```

208

例 6-11 使用自增量方法，为例 6-10 重新编制程序。

解：

```
        ORG    0000H        ;burn into ROM starting at 0
        MOVLW  0x50         ;WREG = 50 low-byte addr
        MOVWF  TBLPTRL      ;look-up table low-byte addr
        MOVLW  0x02         ;WREG = 2, high-byte addr
        MOVWF  TBLPTRH      ;look-up table high-byte addr
        CLRF   TRISB        ;TRSIB = 00 (Port B as output)
B7      TBLRD*+             ;bring in next byte and inc TBLPTR
        MOVF   TABLAT,W     ;copy to WREG (Z = 1, if null)
        BZ     EXIT         ;is it null char? exit if yes
        MOVWF  PORTB        ;send it to Port B
        BRA    B7           ;continue
EXIT    GOTO   EXIT

        ORG    0x250
MYDATA  DB     "USA",0      ;notice null
        END
```

例 6-12 假定从 500H 开始的 ROM 中包含消息 The Promise of World Peace。编程把它读进 CPU，并放到 40H 开始的 RAM 空间，每次一个字节。

解：

```
        ORG    0000H        ;burn into ROM starting at 0
        MOVLW  0x00         ;WREG = 00 low-byte addr
        MOVWF  TBLPTRL      ;look-up table low-byte addr
        MOVLW  0x05         ;WREG = 05, high-byte addr
        MOVWF  TBLPTRH      ;look-up table high-byte addr
        LFSR   2,0x40       ;load pointer. FSR2 = 40H, RAM address
B8      TBLRD*+             ;read the table, then increment TBLPTR
        MOVF   TABLAT,W     ;copy to WREG (Z = 1 if null)
        BZ     EXIT         ;exit if end of string
        MOVWF  POSTINC2     ;copy WREG to RAM and INC FSR2
        BRA    B8
EXIT    GOTO   EXIT

;---------------------message
        ORG    0x500        ;data burned starting at 0x500
MYDATA DB "The Promise of World Peace",0
        END
```

6.3.4 查表和 RETLW 指令

在微控制器的编程中，查表是一个广泛使用的概念。它允许以最少的操作指令来访问频繁使用的表。例如，在某些应用中需要求 0～9 的平方值，则可以用查表法代替平方值的计算，从而可以节约时间。在 PIC 中，要读取表单元素，首先要调用查询表，然后将一个定值数据加到

PCL(程序计数器的低字节)上,作为查询表的索引。在查表返回时,RETLW 指令将读取的表单内容传送到 WREG 中。请看例 6-13 和例 6-14。

例 6-13　假定端口 *C* 的低 3 位已连接上 3 个开关。编制程序,根据 3 个开关的状态把对应的 ASCII 码发到端口 *D*。

000	'0'
001	'1'
010	'2'
011	'3'
100	'4'
101	'5'
110	'6'
111	'7'

解:

```
        ORG    0
        SETF   TRISC         ;TRISC = FFh (Port C as input)
        CLRF   TRISD         ;TRISD = 00 (Port D as output)
B1      MOVF   PORTC,W       ;read x from Port C into WREG
        ANDLW  B'00000111'   ;mask upper 5 bits
        CALL   ASCI_TABLE    ;get ASCII from look-up table
        MOVWF  PORTD         ;copy it to Port D
        BRA    B1            ;continue

;look-up table for ASCII numbers 0-7
ASCI_TABLE
        MULLW  0x2    ;align it for even address for 2-byte RETLW opcode
        MOVFF  PRODL, WREG ;put it into WREG for indexing
        ADDWF  PCL           ;PCL = PCL + WREG
        RETLW  '0'           ;ASCII for 0
        RETLW  '1'           ;ASCII for 1
        RETLW  '2'           ;ASCII for 2
        RETLW  '3'           ;notice that each ASCII value is placed
        RETLW  '4'           ;in the ROM at an even address
        RETLW  '5'
        RETLW  '6'
        RETLW  '7'
        END
```

Program Memory

Address	00	02	04	06	08	0A	0C	0E	ASCII
0000	6894	6A95	5282	0E07	0B07	EC09	F000	6E83	.h.j.R.. ....n
0010	D7FA	0D02	CFF3	FFE8	26F9	0C30	0C31	0C32	....&0.1.2.
0020	0C33	0C34	0C35	0C36	0C37	FFFF	FFFF	FFFF	3.4.5.6. 7.......
0030	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	....

Opcode Hex　Machine　Symbolic

例 6-14　编制程序,从端口 *B* 读入 x 的值,然后将 x^2 发送到端口 C。假定 RB3～RB0 的 x 值为 0～9。要求用查表法代替乘法指令。

如果端口 *B* 的值是 9,那么端口 *C* 的值是多少呢?

解：

```
        ORG    0
        SETF   TRISB          ;TRISB = FFh (Port B as input)
        CLRF   TRISC          ;TRISC = 00 (Port C as output)
B1      MOVF   PORTB,W        ;read x from Port B into WREG
        ANDLW  0x0F           ;mask upper bits
        CALL   XSQR_TABLE     ;get x² from the look-up table
        MOVWF  PORTC          ;copy it to Port C
        BRA    B1             ;continue

;look-up table for square of numbers 0-9
XSQR_TABLE
        MULLW  0x2            ;align it for even address
        MOVFF  PRODL, WREG    ;put it into WREG for indexing
        ADDWF  PCL            ;PCL = PCL + WREG
        RETLW  D'0'           ;square of 0
        RETLW  D'1'           ;square of 1
        RETLW  D'4'           ;square of 2
        RETLW  D'9'           ;square of 3
        RETLW  D'16'          ;square of 4 (10 hex)
        RETLW  D'25'          ;square of 5 (19 hex)
        RETLW  D'36'          ;square of 6 (24 hex)
        RETLW  D'49'          ;square of 7 (31 hex)
        RETLW  D'64'          ;square of 8 (40 hex)
        RETLW  D'81'          ;square of 9 (51 hex)
        END
```

从上面的屏幕截图可以看到，地址 001A 包含指令 RETLW D'0' 的操作码和操作数，即 0 的平方数。地址 001C 的操作数是 01，即 1 的平方数。地址 001E 的操作数是 04，即 2 的平方数。地址 0020 的操作数是 09，即 3 的平方数。地址 0022 的操作数是 10，即 4 的平方数（$4\times4=16=10H$），等等。注意，奇地址的内容是 RETLW 的操作码，即 0x0C。如果端口 B 的值是 9，那么端口 C 的内容为 51H，即 81 的十六进制数（$9^2=81$）。

Program Memory

Address	00	02	04	06	08	0A	0C	0E	ASCII
0000	6893	6A94	5281	0E07	0B0F	EC09	F000	6E82	.h.j.R.. ....n
0010	D7F9	0D02	CFF3	FFE8	26F9	[illegible]	0C01	0C04	....&......
0020	0C09	0C10	0C19	0C24	0C31	0C40	0C51	FFFF	......$. 1.@.Q...
0030	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	....
0040	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	....
0050	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	....
0060	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	....

Opcode Hex | Machine | Symbolic

6.3.5 访问 RAM 中的查询表

查询表也可以放在 RAM 中。有时候需要从 RAM 中读取查询表的内容，因为这些内容是动态的、可修改的。PIC18 允许使用 FSR 作为指针来查询动态表。例如，指令 MOVFF PLUSW2, PORTD 将读取 FSR2＋WREG 所指 RAM 地址的查询表内容。在这里，WREG 用作查询表的索引寄存器。请看例 6-15 和例 6-16。

例 6-15　重做例 6-13，假定在起始地址为 20H 的数据存储区有一个查询表，其内容如下：

20 = ('0')
21 = ('1')
22 = ('2')
23 = ('3')
24 = ('4')
25 = ('5')
26 = ('6')
27 = ('7')

解：

```
        ORG    0
        SETF   TRISC                ;TRISC = FFh (Port C as input)
        CLRF   TRISD                ;TRISD = 00 (Port D as output)
        LFSR   2,0x20               ;load pointer, FSR2 = 20H, RAM address
B1      MOVF   PORTC,W              ;read x from Port C into WREG
        ANDLW  B'00000111'          ;mask upper 5 bits
        MOVFF  PLUSW2,PORTD         ;get data pointed to by FSR2 + WREG
        BRA    B1
        END
```

File Registers

Address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
0000	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0020	30	31	32	33	34	35	36	37	00	00	00	00	00	00	00	00	01234567 ..
0030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

Hex　Symbolic

Special Function Registers

Address	SFR Name	Hex	Decimal	Binary	Char
0F80	PORTA	00	0	00000000	.
0F81	PORTB	00	0	00000000	.
0F82	PORTC	07	7	00000111	.
0F83	PORTD	37	55	00110111	7
0F84	PORTE	00	0	00000000	.

212

在使用 MPLAB 仿真该程序时，请确保 RAM 地址 20H ~ 27H 存有查询表的内容。注意，指令 MOVFF PLUSW2,PORTD 是从 RAM 中读取数据并传送到端口 *D*。

例 6-16　编制程序，从端口 *B* 读入数据 x，然后发送 x^2+2x+3 到端口 *C*。假定 PB3 ~ PB0 有 0 ~ 9 的 x 值。使用查询表的方法代替乘法运算指令。

解：

```
        ORG    0
        SETF   TRISB                ;TRISB = FFh (Port B as input)
        CLRF   TRISC                ;TRISC = 00 (Port C as output)
B1      MOVF   PORTB,W              ;read x from Port B into WREG
        ANDLW  0x0F                 ;mask upper bits
        CALL   XSQR_TABLE           ;get x² from the look-up table
        MOVWF  PORTC                ;copy it to Port C
        BRA    B1                   ;continue
```

```
XSQR_TABLE
        MULLW 0x2           ;align it for even address
        MOVFF PRODL, WREG   ;put it into WREG for indexing
        ADDWF PCL           ;PCL = PCL + WREG
        RETLW D'3'          ;(0)² + 2(0) + 3 = 3
        RETLW D'6'          ;(1)² + 2(1) + 3 = 6
        RETLW D'11'         ;(2)² + 2(2) + 3 = 11
        RETLW D'18'         ;(3)² + 2(3) + 3 = 18
        RETLW D'27'         ;(4)² + 2(4) + 3 = 27
        RETLW D'38'         ;(5)² + 2(5) + 3 = 38
        RETLW D'51'         ;(6)² + 2(6) + 3 = 51
        RETLW D'66'         ;(7)² + 2(7) + 3 = 66
        RETLW D'83'         ;(8)² + 2(8) + 3 = 83
        RETLW D'102'        ;(9)² + 2(9) + 3 = 102
        END
```

6.3.6 PIC18 的写表操作

在 PIC18 中有一条指令 TBLWRT，允许向程序 ROM 中写入数据。指令 TBLRD 用于 PIC18 系列的任何芯片时，不用考虑芯片 ROM 的类型，而指令 TBLWRT 则只能用于带 FlashROM 的 PIC18 芯片。TBLWRT 指令不能工作在 OTP 的 PIC18 上，或者有掩模 ROM 的芯片上。在掩模 ROM 中，程序和数据在 Microchip 公司封装芯片时就已经烧进 ROM 里。由于写 flash ROM 涉及配置位操作，所以这将在第 14 章中讨论。

6.3.7 复习题

1. 指令 TBLRD* 用_______寄存器作为地址指针。
2. 指令 TBLRD* 执行后，哪个寄存器的值加 1?
3. 在 TBLRD* 指令读取数据后，哪个寄存器用来保持数据?
4. 指令 TBLPTR 的大小是多少?它能覆盖多大的 ROM 空间?
5. 指令 TBLRD* + 执行后，哪个寄存器的值加 1?
6. 指令 TBLRD* + 和 TBLRD+ * 之间的差别在哪里?
7. 判断对错：指令 TBLWT 可在 PIC18 系列的任何 ROM 上运行。

6.4 数据 RAM 的位寻址

很多微处理器(如 Intel 386、奔腾)都只允许按字节访问寄存器和 I/O 端口。换而言之，如果想检测 I/O 端口的某个位，那么必须先读端口的整个字节，然后对该字节数进行某些逻辑操作，这样才能得到该位。这种情况在 PIC 单片机中不会遇到，这已在第 4 章提到过。确实，PIC 芯片的重要特点之一就是，既可以按位又可以按字节访问文件寄存器的 RAM 地址。PIC18 的所有 I/O 端口、SFR 和通用寄存器都是可以位寻址的，因为它们都是数据内存里的文件寄存器。WREG 寄存器也可以位寻址，因为它是 SFR。这就是 PIC18 系列芯片的一个很强大的功能。在这一节中，将给出 PIC 系列许多关于位寻址的例子。

6.4.1 可位寻址的文件寄存器数据 RAM

PIC18 的全部 4096 B的 RAM空间都是可以位寻址的。ROM 程序空间只能按字节寻址，而 4 KB的数据 RAM 则既可以按字节寻址，又可以按位寻址。为了区别数据 RAM 中的按字节寻址和按位寻址，PIC18 提供了两类指令：面向位的指令和面向字节的指令。面向位的指令被称为位寻址，而面向字节的指令被称为字节寻址。

面向位的指令如表 6-4 所示。注意，面向位的指令只能使用一种寻址方式，即直接寻址方式。在本章的前 3 节中已经介绍了 PIC18 可字节寻址空间的几种寻址方式。其中，寄存器间接寻址方式既可以用于数据 RAM，又可以用于程序 ROM。注意，对于 PIC 的面向位的指令是不存在寄存器间接寻址方式的。

表 6-4　PIC18 的位寻址(面向位的) 指令

指　令	功　能
BSF fileReg,bit	置位 fileReg（置位：bit = 1）
BCF fileReg,bit	清零 fileReg（清零：bit = 0）
BTG fileReg,bit	取反 fileReg（取反：bit = $\overline{\text{bit}}$）
BTFSC fileReg,bit	位测试 fileReg，若清零则跳过（若 bit = 0，则跳过下一条指令）
BTFSS fileReg,bit	位测试 fileReg，若置位则跳过（若 bit = 1，则跳过下一条指令）

注意：fileReg 可以是文件寄存器数据 RAM 的任意位置。

6.4.2 文件寄存器的位寻址

在第 2 章已经讨论到，PIC18 系列根据芯片型号的不同，有最多达 4096 B 的文件寄存器数据 RAM，既可以访问文件寄存器的 8 个位，也可以访问其中的某个位而不影响其他的位。访问文件寄存器的某个位时，可以使用 Bit- Oriented- instr fileReg,x 指令格式，其中 fileReg 是任意的文件寄存器，x 是要操作的位的序号 0～7，即分别代表 D0～D7。例如，指令 BTG 0x20,7 的作用是对 RAM 中地址 20H 的 D7 位取反。正如在本章的前面部分提到的，包括 WREG 在内的每个寄存器在文件寄存器中都拥有一个 8 位的地址，而端口 PORTA～PORTE 也是文件寄存器的组成部分。例如，指令 BSF PORTB,5 把端口 B 的 RB5 位置为高电平。注意，当对指令 BSF PORTB,5 汇编时，它将变成 8A81，因为端口 B 在访问存储区的 RAM 地址是 81H。研究下面的几个例子可以深入理解 PIC18 文件寄存器的位寻址。

214

例 6-17　RC7 引脚已连接上一个开关。编制程序，查询开关的状态，并根据状态执行下列操作：

(1) 若为 0，则发送'N' 到端口 D；

(2) 若为 1，则发送'Y' 到端口 D。

解：

```
       BSF   TRISC,7      ;make RC7 an input
       CLRF  TRISD        ;make Port D an output port
AGAIN  BTFSS PORTC,7      ;bit test RC7 for HIGH
       BRA   OVER         ;it must be LOW
       MOVLW A'Y'         ;WREG = 'Y' ASCII letter Y
       MOVWF PORTD        ;issue WREG to PD
```

```
        GOTO   AGAIN        ;we could use BRA instead
OVER    MOVLW  A'N'         ;WREG = 'N' ASCII letter N
        MOVWF  PORTD        ;issue WREG to PORTD
        GOTO   AGAIN        ;we can use BRA too
```

例 6-18 编制程序，对 RB1 取反 200 次。用文件寄存器 RAM 地址 32H 保存计数器的值。

解：

```
        MYREG EQU 0x32      ;set aside loc 0x20 reg
        CNTVAL EQU D'200'
        MOVLW CNTVAL        ;load counter into WREG
        MOVWF MYREG         ;load the count into MYREG location
        BCF    TRISB,1      ;TRISB bit = 0, make RB1 an output
AGAIN   BTG    PORTB,1      ;toggle bit RB1
        DECF   MYREG,F      ;decrement MYREG
        BNZ    AGAIN        ;continue until counter is zero
```

215

例 6-19 引脚 RC7 已连到一个开关上。编制程序，查询开关的状态并根据状态执行以下操作：

(a) 如果 RC7 = 0，将端口 *B* 加 1；

(b) 如果 RC7 = 1，将端口 *B* 减 1。

解：

```
        BSF    TRISC,7      ;make RC7 an input
        CLRF   TRISB        ;make Port B an output port
AGAIN   BTFSS  PORTC,7      ;bit test RC7 for HIGH
        BRA    OVER         ;it must be LOW
        INCF   PORTB,F      ;increment
        GOTO   AGAIN        ;we can use BRA too
OVER    DECF   PORTB,F      ;decrement
        GOTO   AGAIN        ;we can use BRA too
```

例 6-20 将引脚 RB0 连到一个开关上。编制程序，查询开关的状态并保存到文件寄存器 0x20 的 D0 位。

解：

```
MYBITREG EQU 0x20           ;set aside loc 0x20 reg
        BSF    TRISB,0      ;make RB0 an input
AGAIN   BTFSS  PORTB,0      ;bit test RB0, skip if set
        GOTO   OVER         ;it must be LOW (BRA is OK too)
        BSF    MYBITREG,0   ;set bit D0 = 1
        GOTO   AGAIN        ;we can use BRA too
OVER    BCF    MYBITREG,0   ;clear D0 (D0 = 0)
        GOTO   AGAIN        ;we can use BRA too
```

例 6-21 编制程序，检查 RAM 中 37H 位置是否包含了一个偶数。如果是，那么传送到端口 *B*；如果不是，把它变成偶数，然后传送到端口 *B*。

解：

```
MYREG EQU 0x37              ;set aside loc 0x37 reg
        CLRF   TRISB        ;make Port B an output port
AGAIN   BTFSS  MYREG,0      ;bit test D0, skip if set
        GOTO   OVER         ;it must be LOW
        BCF    MYREG,0      ;clear bit D0 = 0
OVER    MOVFF  MYREG,PORTB  ;copy it to Port B
        GOTO   AGAIN        ;we can use BRA too
```

216

例 6-22 编制程序，使端口 *B* 计数：从 0000 计数到 1111(二进制)。

解：

```
      CLRF  TRISB       ;TRISB = 0, make PB output
      CLRF  PORTB       ;Port B = 0
AGAIN INCF  PORTB,F     ;increment Port B
      BTFSS PORTB,4     ;test D4 bit of Port B
      BRA   AGAIN
      GOTO  $
```

注意程序是怎样从 0000 数到 1111 的。当计数值变为 10000 时，跳出循环。

例 6-23 编制程序，检查文件寄存器 0x20 位置的 D7 位的状态，然后：

(a) 若 D7 = 0，则传送 NO 到端口 *B*；

(b) 若 D7 = 1，则传送 YES 到端口 *B*。

解：

```
MYREG EQU  0x20
      CLRF  TRISB       ;make Port B an output port
AGAIN BTFSS MYREG,7     ;bit test for HIGH
      BRA   OVER        ;it must be LOW
      MOVLW A'Y'        ;WREG = 'Y' ASCII letter Y
      MOVWF PORTB       ;issue WREG to Port B
      MOVLW A'E'        ;WREG = 'E' ASCII letter Y
      MOVWF PORTB       ;issue WREG to Port B
      MOVLW A'S'        ;WREG = 'S' ASCII letter Y
      MOVWF PORTB       ;issue WREG to Port B
      GOTO  AGAIN       ;we can use BRA too
OVER  MOVLW A'N'        ;WREG = 'N' ASCII letter N
      MOVWF PORTB       ;issue WREG to Port B
      MOVLW A'O'        ;WREG = 'O' ASCII letter Y
      MOVWF PORTB       ;issue WREG to Port B
      GOTO  AGAIN       ;we can use BRA too
```

6.4.3 状态寄存器的位寻址

在具有位寻址能力的寄存器中，首先关注状态寄存器。其余的寄存器将在接下来的章节中讨论。

217

现在让我们来看看怎样使用状态寄存器的位寻址。正如第 2 章讨论的，状态寄存器的 5 个位分别表示标志 C、DC、Z、N 和 OV。请看图 6-2 和例 6-24。

D7							D0
X	X	X	N	OV	Z	DC	C

C：进位标志位　　OV：溢出标志位

DC：数字进位标志位　　N：负数标志位

Z：零标志位

X：D5、D6和D7位预留

图 6-2 状态寄存器的各位

例 6-24 当使用指令 BC 和 BZ 来查询进位和零标志位时，请说明是如何使用状态寄存器标志来检测进位标志位和零标志位的。

解：

(a) C 标志位是状态寄存器的 D0 位，因此可以使用下面的指令来检查 C 标志位：

```
BTFSS STATUS,C     ;bit test C, skip if C = 1
```

(b) Z 标志位是状态寄存器的 D3 位，因此可以使用下面的指令来检查 Z 标志位：

```
BTFSS STATUS,Z     ;bit test Z, skip if Z = 1
```

6.4.4 复习题

1. 判断对错：PIC18 的所有 I/O 端口都是可以位寻址的。
2. 判断对错：PIC18 的状态寄存器是可以位寻址的。
3. 判断对错：PIC18 的所有文件寄存器 RAM 位置都是可以位寻址的。
4. 指出下列寄存器中哪些是可以位寻址的：
   (a) 端口 *A* (b) 端口 *B* (c) WREG (d) 状态寄存器 (e) 21 位 PC
5. PIC18 的 4096 B 的 RAM 中，有多少字节是可以位寻址的？
6. 你将会怎样检查 RAM 中地址 3 的 D1 位是高电平还是低电平？
7. 请说明每条指令的作用。
   (a) `BSF 0x20,1` (b) `BCF 0x32,7` (c) `BSF 0x12,7`
   (d) `BSF PORTB,4` (e) `BSF STATUS,1`
8. 请说明怎样清零进位标志位。

218

6.5 PIC18 的存储区转换

PIC18 微控制器有最多达 4 KB 的数据存储空间。虽然各款 PIC18 芯片的内存大小不同，但它们至少都有用于文件寄存器的访问存储区。文件寄存器 RAM 被分成 256 B 大小的存储区，因此 PIC18 总共有 16 个存储区。正如在第 2 章所讨论的，每片 PIC18 都带有的最小的存储区，又被称作访问存储区。访问存储区由低地址和高地址组成，各占 128 B。低地址的 128 B(000 ～ 07FH) 用于通用寄存器 RAM，而高地址的 128 B(F80 ～ FFFH) 用于 SFR。在 Microchip 网站上看到的大多数 PIC18 芯片都带有除访问存储区之外的存储区。本节将讨论如何使用存储区转换技术来利用 PIC18 的全部 RAM 空间。

6.5.1 位 *A* 和存储区转换

到现在为止，所用到的所有指令都是假定访问存储区为默认的存储区。这是通过忽略指令中的位 *A* 来实现的，如 `MOVWF fileReg, A`。换而言之，指令 `MOVWF fileReg` 实际上就是 `MOVWF fileReg,A`，其中位 *A* 可以是 0 或 1。若 $A=0$，则访问存储区是默认存储区。若 $A=1$，则该指令使用 BSR 指定的存储区，而不是访问存储区。若在指令中没有出现位 *A* 的声明，则表示 $A=0$，访问存储区就是默认的存储区。到现在为止这么做是为了使 PIC18 的汇编语言更易懂、更易掌握。接下来将讨论 BSR 寄存器在存储区转换中的作用。

6.5.2 BSR寄存器和存储区转换

如果要使用其他的存储区，那么需要在指令条件中将 A 置 1，同时使用 BSR(存储区选择寄存器)来选择期望的存储区。BSR是一个8位的寄存器，属于SFR。在BSR的8位中，只有低4位用于PIC18，高4位置0并被PIC18忽略。BSR的低4位确定了16个存储区，每个存储区包含256 B，因此可以使用存储区转换技术来遍历数据 RAM 文件寄存器的 4096 B。4 KB的数据RAM被分成存储区0～F，最低的存储区0的地址是00～FFH，最高的存储区F的地址是F00～FFFH。在PIC18中，存储区F的最后128 B通常被留作SFR。通用寄存器通常从存储区0的地址00开始。在上电复位时，BSR＝0(0000B)，这表明除SFR(F80～FFFH)以外，只有数据RAM的低地址(000～0FFH)可用作通用寄存器。类似地，如果BSR＝1 (0001B)，那么PIC18选择存储区1，使用地址段(100～1FFH)和SFR地址(F80～FFFH)。若想要使用存储区2，则必须传送02(0010B)到BSR中，这样就可以使用存储区地址(200～2FFH)和SFR地址(F80～FFFH)了。不管PIC18数据RAM空间的大小，通用寄存器总是从地址000开始的，而且是递增的，而SFR则从地址FFF开始，而且是递减的。目前，PIC只使用了存储区F的高128 B(F80～FFFH)作为SFR。或许在不远的将来，随着更多的特殊功能嵌入到PIC18中，将会用到存储区F的剩余部分，甚至用到存储区E来作为SFR。虽然PIC18芯片根据嵌入芯片的功能不同，用于SFR的存储区F的字节数可能会有所不同，但是SFR通常都是从地址FFF开始，并且是递减的。这是必须要强调的一点。例如，就访问存储区来讲，存储区F的后半部分空间都预留为SFR，尽管并不是所有PIC型号都支持128种特殊功能。表6-5列出了一些PIC18芯片的数据RAM文件寄存器。注意，虽然可以使用任何寻址方式(如立即寻址、直接寻址、寄存器间接寻址)来访问通用寄存器区，但是却只能使用直接寻址方式来访问SFR。为了更好地理解存储区转换技术，下面以PIC18F458为例给出一些例子。

219

表 6-5 部分 PIC18 芯片的数据 RAM 存储区

	文件寄存器 (B)	=	SFR (B)	+	用作 GPR 的可用空间 (B)
PIC18F1220	512		256		256
PIC18F452	1792		256		1536
PIC18F2220	768		256		512
PIC18F458	1792		256		1536
PIC18F8722	4096		158		3938

220

注意：最新版本的 PIC18F458/452 是 PIC18F4580/4520。摘自 http://www.microchip.com。

6.5.3 存储区转换和指令INCF F,D,A

PIC18F458 总共有 1792 B的数据 RAM 文件寄存器。PIC18F458 的存储区结构如图 6-3 所示。到现在为止，所举的例子都忽略了位 A，即默认为 A＝0，访问存储区成为默认存储区。若要使用其他的存储区，则必须执行下面的两个步骤：

(1) 向 BSR 写入期望的存储区序号；

(2) 在指令自身中将 A 置 1。

因此，指令 INCF MYREG,F,1 与 INCF MYREG,F,0 是截然不同的。A＝1 表示使用 BSR 所指的

存储区。在下面的代码中，首先用指令 MOVLB将存储区序号写入到BSR中，然后对RAM地址0x240(存储区 2 的 40H 的)的内容进行操作。

```
MYREG EQU    0x40

MOVLB 0x2          ;load 2 into BSR (use bank 2)
MOVLW 0            ;WREG = 0
MOVWF MYREG, 1     ;loc 0x240 = (0), WREG = 0, Notice A = 1
INCF  MYREG, F, 1 ;loc 0x240 = (1), WREG = 0, Notice A = 1
INCF  MYREG, F, 1 ;loc 0x240 = (2), WREG = 0, Notice A = 1
INCF  MYREG, F, 1 ;loc 0x240 = (3), WREG = 0
```

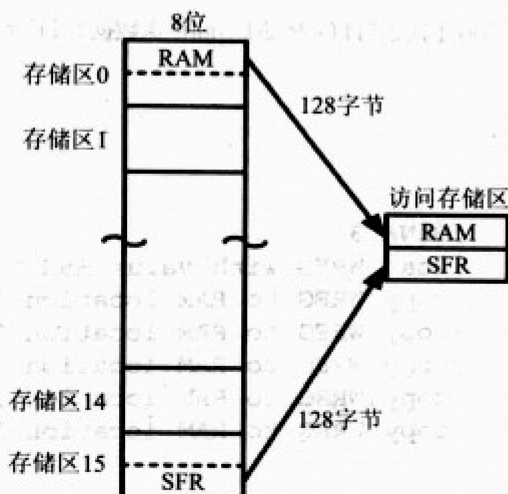


图 6-3　数据 RAM 寄存器

将上面的程序同下面的程序相对照：

```
MOVLB 0x2          ;load 2 into BSR (use bank 2)
MOVLW 0            ;WREG = 0
MOVWF MYREG        ;loc 0x40 = (0), WREG = 0
INCF  MYREG, F     ;loc 0x40 = (1), WREG = 0, Notice A = 0
INCF  MYREG, F     ;loc 0x40 = (2), WREG = 0, Notice A = 0
INCF  MYREG, F     ;loc 0x40 = (3), WREG = 0
```

尽管已经向 FSR 赋值，但是因为位 *A* 在指令中没有被声明，所以 MPASM 同样默认它为0，即使用访问存储区的地址 0x40。指令域中的位 *A* 如图 6-4 所示。



D=F，目标地址是fileReg
D=W，目标地址是WREG

D—操作的目标地址
A—操作需要访问的存储区
0≤f≤FF

A=0，使用默认的访问存储区
A=1，使用BSR所指的存储区

图 6-4　INCF 指令格式中的位 *A*

考察下面的代码，分析位 *D* 和位 *A* 的作用。

```
MOVLB 0x2           ;load 2 into BSR (use bank 2)
MOVLW 0             ;WREG = 0
MOVWF 0x20,1        ;loc 0x220 = (0), WREG = 0, D = W, A = 1 means Bank 2
INCF  0x20,W,1      ;loc 0x220 = (0), WREG = 1, D = W, A = 1
INCF  0x20,W,1      ;loc 0x220 = (0), WREG = 1, D = W, A = 1
INCF  0x20,W,1      ;loc 0x220 = (0), WREG = 1, D = W, A = 1
INCF  0x20,F,1      ;loc 0x220 = (1), WREG = 1, D = F, A = 1
INCF  0x20,F,1      ;loc 0x220 = (2), WREG = 1, D = F, A = 1
INCF  0x20,F,1      ;loc 0x220 = (3), WREG = 1, D = F, A = 1
INCF  0x20,F,1      ;loc 0x220 = (4), WREG = 1, D = F, A = 1
```

用MPLAB仿真下面的例子,查看数据RAM,观察存储区转换的实现。

例6-25　编制程序,向从340H到345H的RAM空间传送数据55H。要求使用:

(a) 直接寻址方式;

(b) 循环。

解:

(a)

```
MOVLB 0x3           ;BANK 3
MOVLW 0x55          ;load WREG with value 55H
MOVWF 0x40, 1       ;copy WREG to RAM location 340H
MOVWF 0x41, 1       ;copy WREG to RAM location 341H
MOVWF 0x42, 1       ;copy WREG to RAM location 342H
MOVWF 0x43, 1       ;copy WREG to RAM location 343H
MOVWF 0x44, 1       ;copy WREG to RAM location 344H
```

(b)

```
COUNT EQU 0x10              ;loc 10h
      MOVLB  0x3            ;BANK 3
      MOVLW  0x5            ;WREG = 5
      MOVWF  COUNT          ;load the counter, count = 5
      LFSR   0,0x340        ;load pointer. FSR0 = 40H, RAM address
      MOVLW  0x55           ;WREG = 55h value to be copied
B1    MOVWF  INDF0,0        ;copy WREG to RAM loc FSR0 points to
      INCF   FSR0L          ;increment FSR0L pointer
      DECF   COUNT,F,0      ;decrement the counter
      BNZ    B1             ;loop until counter = zero
```

程序执行后,RAM地址的内容为:

340 = (55)

341 = (55)

342 = (55)

343 = (55)

344 = (55)

例6-25的数据RAM的内容如图6-5所示。

表6-6列出了PIC18芯片中不同数据RAM空间的存储区。

File Registers

Address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
0320	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0330	00	00	00	00	00	00	00	00	00	[illegible]	00	00	00	00	00	00	....
0340	55	55	55	55	55	00	00	00	00	[illegible]	00	00	00	00	00	00	UUUUU... ....
0350	00	00	00	00	00	00	00	00	00	[illegible]	00	00	00	00	00	00	....
0360	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0370	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

Hex Symbolic

图 6-5 例 6-25 的数据 RAM 内容

表 6-6 PIC18 数据存储器范围

数据存储器	存 储 区	数据存储器	存 储 区
64	0,15	2048	0～7,15
128	0,15	2304	0～8,15
256	0,15	2560	0～9,15
512	0～1,15	2816	0～10,15
640	0～2,15	3072	0～11,15
768	0～2,15	3328	0～12,15
1024	0～3,15	3584	0～13,15
1280	0～4,15	3840	0～14,15
1536	0～5,15	3968	0～15
1792	0～6,15		

6.5.4 MOVFF 指令和存储区

使用指令 *MOVFF* 的很大好处就是不用考虑存储区的转换，因为它可以在 4 *KB RAM* 空间内传送数据。请参阅图 6-4 和例 6-26。

例 6-26 假定 PIC18F458 的 RAM 地址 330～334H 中有如下的 ASCII 字符串。编制程序，读取每个字符，然后将它们发送到端口 B，每次发送一个字节。在编程时请使用：

(a) 直接寻址方式；

(b) 寄存器间接寻址方式。

330 = ('H')

331 = ('E')

332 = ('L')

333 = ('L')

334 = ('O')

解：

(a) 直接寻址方式：

```
CLRF  TRISB              ;make Port B an output
MOVFF 0x330, PORTB       ;copy contents of loc 0x330 to PB
MOVFF 0x331, PORTB
MOVFF 0x332, PORTB
MOVFF 0x333, PORTB
MOVFF 0x334, PORTB
```

(b)

```
COUNTREG EQU 0x20                  ;fileReg loc 20 for counter
CNTVAL EQU 5                       ;counter value
        CLRF  TRISB                ;make Port B an output (TRSIB = out)
        MOVLW CNTVAL               ;WREG = 5
        MOVWF COUNTREG             ;load the counter, count = 5
        LFSR  2,0x330              ;load pointer. FSR2 = 330H, RAM address
B3      MOVF  INDF2,W              ;copy RAM loc FSR2 points at to WREG
        MOVWF PORTB                ;copy WREG to PORTB
        INCF  FSR2L                ;increment FSR2 to point at next loc
        DECF  COUNTREG,F           ;decrement counter
        BNZ   B3                   ;loop until counter = zero
```

File Registers

Address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
02F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0300	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0310	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0320	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0330	48	65	6C	6C	6F	00	00	00	00	00	00	00	00	00	00	00	Hello... ....
0340	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

Hex　Symbolic

图 6-6　在 MPLAB 中例 6-26 的数据 RAM 内容

6.5.5　用 MPLAB 仿真器检查数据 RAM 空间

223～224

MPLAB 仿真器在检查数据 RAM 的内容方面是一个很好的工具。应该鼓励使用 MPLAB 来查看和校验使用数据 RAM 的程序执行的结果。

例 6-27　为 PIC18F452 芯片编制程序，传送数据 FFH 到起始地址为 160H 的 16 个 RAM 地址中。要求使用：

(a) INCF FSRnL 指令；

(b) 自增量方式。

解：

(a)

```
COUNTREG EQU 0x10          ;fileReg loc for counter
CNTVAL EQU D'16'           ;counter value
        MOVLW CNTVAL       ;WREG = 16
        MOVWF COUNTREG     ;load the counter, count = 16
        LFSR  1,0x160      ;load pointer. FSR1 = 60H, RAM address
        MOVLW 0xFF         ;load 0xFF
B2      MOVWF INDF1,0      ;move W to RAM loc FSR1 points to
        INCF  FSR1L        ;increment FSR1L, point to next loc
        DECF  COUNTREG,F   ;decrement counter
        BNZ   B2           ;loop until counter = zero
```

(b)

```
COUNTREG EQU 0x10          ;fileReg loc for counter
CNTVAL EQU D'16'           ;counter value
        MOVLW CNTVAL       ;WREG = 16
```

```
        MOVWF COUNTREG      ;load the counter, count = 16
        LFSR  1,0x160       ;load pointer. FSR1 = 160H, RAM address
        MOVLW 0xFF          ;load 0xFF
B3      MOVWF POSTINC1, 0
        DECF  COUNTREG,F    ;decrement counter
        BNZ   B3            ;loop until counter = zero
```

File Registers

Address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
0140	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0150	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0160	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
0170	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0180	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0190	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

Hex Symbolic

图 6-7　例 6-27 的数据 RAM 内容

例 6-28　编制程序,将 RAM 地址 330H ~ 33FH 存放的数据块复制到地址 360H ~ 36FH 中。

解:

```
COUNTREG EQU 0x20             ;fileReg loc 20 for counter
CNTVAL EQU 0x0F               ;counter value = 15
        CLRF  TRISB           ;make Port B an output (TRSIB = FFH)
        MOVLW CNTVAL          ;WREG = 15
        MOVWF COUNTREG,1      ;load the counter, count = 15
        LFSR  1,0x330         ;load pointer. FSR1 = 330H, RAM address
        LFSR  2,0x360         ;load pointer. FSR2 = 360H, RAM address

B3      MOVFF INDF1, INDF2
        INCF  FSR1L           ;increment FSR1 to point at next loc
        INCF  FSR2L           ;increment FSR2 to point at next loc
        DECF  COUNTREG,F      ;decrement counter
        BNZ   B3              ;loop until counter = zero
```

File Registers

Address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
0320	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0330	48	65	6C	6C	6F	20	57	6F	72	6C	64	21	00	00	00	00	Hello Wo rld!....
0340	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0350	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0360	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0370	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

Hex Symbolic

File Registers

Address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
0320	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0330	48	65	6C	6C	6F	20	57	6F	72	6C	64	21	00	00	00	00	Hello Wo rld!....
0340	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0350	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0360	48	65	6C	6C	6F	20	57	6F	72	6C	64	21	00	00	00	00	Hello Wo rld!....
0370	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

Hex Symbolic

图 6-8　例 6-28 程序运行前后的文件寄存器的内容

6.5.6 复习题

1. 判断对错:PIC18 使用最后一个存储区作为 SFR。
2. 判断对错:PIC18 每个存储区都是 256 B。
3. 判断对错:PIC18 的前 128 B RAM 用作访问存储区。
4. 请给出用于 SFR 的更高位地址。
5. 在 PIC18F458 中,怎样将数据 99H 传送到 RAM 地址 202H?
6. 在带有 4 KB RAM 的 PIC18 中,怎样将数据 55H 传送到 RAM 地址 408H?
7. 判断对错:在 PIC18 中,指令 MOVFF 可以在 RAM 空间里任意地复制字节数据。
8. BSR 是________位寄存器,但只有________位是用于选择存储区的。

226

6.6 校验和与 ASCII 码子例程

本节将介绍一些广泛使用的子例程,如校验和、BCD 和 ASCII 码的转换。同时,也会给出 PIC18 栈的应用。

6.6.1 ROM 中的校验和

为了保证 ROM 内容的完整性,每个系统都必须执行校验和计算。校验和可以检测出 ROM 中内容是否遭到破坏。在系统打开或者系统运行时,浪涌电流的冲击可能会引起 ROM 内容的破坏。为保证 ROM 中数据的完整性,校验和过程使用了一个校验和字节数。校验和字节数是紧跟在数据块之后的额外字节。要计算一串数据的校验和,请遵循下面的步骤。

(1) 将所有的字节数据相加,丢弃进位。

(2) 对得到的代数和做取补运算。这就是校验和字节数,放在字节数据块的末尾。

在执行校验和操作时,将所有的字节数据相加(包括校验和字节数)。正常情况下,所得的结果为 0。若不为 0,则表明有的数据已被破坏(改变)。为了更好理解这些重要的概念,请参阅例 6-29。

6.6.2 校验和程序

校验和的产生程序和测试程序以子例程的形式给出。因此,可以把程序 6-1 分解成 3 个子例程,分别执行下列的操作。

(1) 从程序 ROM 中读取数据。

(2) 计算校验和字节数。

(3) 测试校验和字节,检查数据错误。

每个子例程都可用在其他的应用中。例 6-29 说明如何手工计算一串数据的校验和。也可参阅程序 6-1。

227

例 6-29 假定有 4 个 16 进制的数据:25H、62H、3FH 和 52H。

(a) 计算它们的校验和字节数。

(b) 执行校验和操作,验证数据完整性。

(c) 如果第二个字节62H修改为22H,请说明如何使用校验和方法来诊断错误。

解:

(a) 计算校验和字节数。

```
   25H
+  62H
+  3FH
+  52H
  118H
```

(在丢弃进位1后,得到18H。做取补运算后,得到E8H,即为校验和字节数。)

(b) 执行校验和操作,验证数据完整性。

```
   25H
+  62H
+  3FH
+  52H
+  E8H
  200H
```

(在丢弃进位1后,得到00。这说明数据没有错误。)

(c) 如果第二个字节62H修改为22H,请说明如何使用校验和方法来诊断错误。

```
   25H
+  22H
+  3FH
+  52H
+  E8H
  1C0H
```

(在丢弃进位1后,得到的是C0H,而不是00H。这说明数据出现错误。)

程序6-1 计算和测试校验和字节数

```
;PROG 6-1: CALCULATING AND TESTING CHECKSUM BYTE
        #include P18F458.inc

RAM_ADDR    EQU 40H      ;RAM space to place the bytes
COUNTREG    EQU 0x20     ;fileReg loc for counter
CNTVAL      EQU 4        ;counter value = 4 for adding 4 bytes
CNTVAL1     EQU 5        ;counter value = 5 for adding 5 bytes
                         ;including checksum byte

;------------main program
        ORG     0
        CALL    COPY_DATA
        CALL    CAL_CHKSUM
        CALL    TEST_CHKSUM
        BRA     $

;--------copying data from code ROM address 500H to data RAM loc
COPY_DATA
        MOVLW low(MYBYTE)        ;WREG = 00 LOW-byte addr
        MOVWF TBLPTRL            ;ROM data LOW-byte addr
        MOVLW hi(MYBYTE)         ;WREG = 5, HIGH-byte addr
        MOVWF TBLPTRH            ;ROM data HIGH-byte addr
        MOVLW upper(MYBYTE)      ;WREG = 00 upper-byte addr
        MOVWF TBLPRTRU           ;ROM data upper-byte addr
        LFSR  0,RAM_ADDR         ;FSR0 = RAM_ADDR, place to save
C1      TBLRD*+                  ;bring in next byte and inc TBLPTR
```

```
      MOVF  TABLAT,W            ;copy to WREG (Z = 1, if null)
      BZ    EXIT                ;is it null char? exit if yes
      MOVWF POSTINC0            ;copy WREG to RAM and inc pointer
      BRA   C1
EXIT  RETURN

    ;-----calculating checksum byte
    CAL_CHKSUM
          MOVLW CNTVAL              ;WREG = 4
          MOVWF COUNTREG            ;load the counter, count = 4
          LFSR  0,RAM_ADDR          ;load pointer. FSR0 = 40H
          CLRF  WREG
    C2    ADDWF POSTINC0,W          ;add RAM to WREG and increment FSR0
          DECF  COUNTREG,F          ;decrement counter
          BNZ   C2                  ;loop until counter = zero
          XORLW 0xFF                ;1's comp
          ADDLW 1                   ;2'compl
          MOVWF POSTINC0
          RETURN

    ;----------testing checksum byte
    TEST_CHKSUM
          MOVLW CNTVAL1             ;WREG = 5
          MOVWF COUNTREG            ;load the counter, count = 5
          CLRF  TRISB               ;PORTB = output
          LFSR  0,RAM_ADDR          ;load pointer. FSR0 = 40H
          CLRF  WREG
    C3    ADDWF POSTINC0,W          ;add RAM and increment FSR0
          DECF  COUNTREG,F          ;decrement counter
          BNZ   C3                  ;loop until counter = zero
          XORLW 0x0                 ;EX-OR to see if WREG = zero
          BZ    G_1                 ;is result zero? then good
          MOVLW 'B'
          MOVWF PORTB               ;if not, data is bad
          RETURN
    G_1   MOVLW 'G'
          MOVWF PORTB               ;data is not corrupted
          RETURN

    ;----------my data in program ROM
          ORG 0x500
    MYBYTE DB 0x25, 0x62, 0x3F, 0x52, 0x00
          END
```

229 注意关键字 low、hi 和 upper 的使用，它们是用来提取程序 ROM 的 21 位地址的。

6.6.3 BCD 到 ASCII 的转换程序

许多RTC(实时时钟)都提供BCD格式的时间和日期。为了在LCD或PC屏幕上显示BCD数据，就必须要将BCD数转换成ASCII码。程序6-2的第一部分把程序ROM中的压缩BCD数据传送到数据RAM中，第二部分则把压缩BCD码转换成ASCII码，而第三部分再把ASCII码传送到端口B显示。这里的程序有一部分将在第16章中使用到，关于在LCD上显示数据则将在第12章介绍。BCD到ASCII码的转换算法请参阅第5章。

程序 6-2 从压缩 BCD 到 ASCII 的转换

```
;PROG 6-2: CONVERTING PACKED BCD TO ASCII
      #include P18F458.inc

RAM_ADDR      EQU 0x40
ASC_RAM       EQU 0x50
COUNTREG      EQU 0x20          ;fileReg loc for counter
CNTVAL        EQU D'4'          ;counter value of BCD bytes
CNTVAL1       EQU D'8'          ;counter value of ASCII bytes

;------------main program
      ORG    0
      CALL   COPY_DATA
      CALL   BCD_ASC_CONV
      CALL   DISPLAY
      BRA    $

;--------copying data from code ROM to data RAM
COPY_DATA
      MOVLW low(MYBYTE)         ;WREG = 00 LOW-byte addr
      MOVWF TBLPTRL             ;ROM data LOW-byte addr
      MOVLW hi(MYBYTE)          ;WREG = 5, HIGH-byte addr
      MOVWF TBLPTRH             ;ROM data HIGH-byte addr
      MOVLW upper(MYBYTE)       ;WREG = 00 upper-byte addr
      MOVWF TBLPRTRU            ;ROM data upper-byte addr
      LFSR  0,RAM_ADDR          ;FSR0 = RAM_ADDR, place to save
C1    TBLRD*+                   ;bring in next byte and inc TBLPTR
      MOVF  TABLAT,W            ;copy to WREG (Z = 1, if null)
      BZ    EXIT                ;is it null char? exit if yes
      MOVWF POSTINC0            ;copy WREG to RAM and inc pointer
      BRA   C1
EXIT  RETURN

;-----convert packed BCD to ASCII
BCD_ASC_CONV
      MOVLW CNTVAL              ;get the counter value
      MOVWF COUNTREG            ;load the counter
      LFSR  0,RAM_ADDR          ;FSR0 = RAM_ADR BCD byte pointer
      LFSR  1,ASC_RAM           ;FSR1 = ASC_RAM ASCII byte pointer
B2    MOVF  INDF0,W             ;copy BCD to WREG
      ANDLW 0x0F                ;mask the upper nibble (W = 09)
      IORLW 0x30                ;make it an ASCII
      MOVWF POSTINC1            ;copy to RAM and increment FSR1
      MOVF  POSTINC0,W          ;note the use of instruction
      ANDLW 0xF0                ;mask the lower nibble (W = 20H)
      SWAPF WREG
      IORLW 0x30                ;make it an ASCII
      MOVWF POSTINC1            ;copy to RAM and increment FSR1
      DECF  COUNTREG,F          ;decrement counter
      BNZ   B2                  ;loop until counter = zero
      RETURN

;-----send ASCII data to port B
DISPLAY
      CLRF  TRISB               ;make PORTB output (TRSIB = FFH)
      MOVLW CNTVAL1             ;WREG = 8, send 8 bytes of data
      MOVWF COUNTREG            ;load the counter, count = 8
      LFSR  2,ASC_RAM           ;load pointer. FSR2 = 50H
B3    MOVF  POSTINC2,W          ;copy RAM to WREG and inc pointer
      MOVWF PORTB               ;copy WREG to PORTB
```

```
        DECF  COUNTREG,F          ;decrement counter
        BNZ   B3                  ;loop until counter = zero
        RETURN

;----------my BCD data in program ROM
        ORG  0x500
MYBYTE DB  0x25, 0x67, 0x39, 0x52, 0x00
        END
```

File Registers

Address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
0000	00	52	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.R...... ....
0010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0040	25	67	39	52	00	00	00	00	00	00	00	00	00	00	00	00	%g9R.... ....
0050	35	32	37	36	39	33	32	35	00	00	00	00	00	00	00	00	52769325 ....
0060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

Hex Symbolic

图 6-9 程序 6-2 执行之后的结果

6.6.4 二进制(十六进制)到 ASCII 的转换程序

很多 ADC(模数转换器)芯片都是二进制输出的。为了在 LCD 或 PC 屏幕上显示这些数据，就需要将二进制数转换为 ASCII 数。实现二进制到 ASCII 转换的程序代码如程序 6-3 所示。注意，第一个子例程将从端口 *B* 读入 8 位数据，然后将它转换成十进制数据；而第二个子例程则把十进制数据转换成 ASCII 码并保存。这里将低位数据存放到低地址，把高位数据存放到高地址。这就是所谓的 little-endian 约定：低字节存放到低地址，高字节存放到高地址。所有的 PIC18 产品都采用 little-endian 约定。关于二进制到 ASCII 的转换算法请参阅第 5 章。

File Registers

Address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
0000	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0030	08	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0040	38	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	80000000 00000000

Hex Symbolic

图 6-10 程序 6-3 执行后的结果(请注意文件寄存器 0030H、0040H、0041H 的内容)

程序 6-3 从二进制(十六进制)到 ASCII 的转换

```
;PROG 6-3: CONVERTING BIN(HEX) TO ASCII
#include P18F458.INC

NUME        EQU   0x00        ;RAM loc for NUME
QU          EQU   0x20        ;RAM loc for quotient
RMND_L      EQU   0x30        ;the least significant digit loc
RMND_M      EQU   0x31        ;the middle significant digit loc
RMND_H      EQU   0x32        ;the most significant digit loc
```

```
MYDEN        EQU   D'10'          ;value for divide by 10

COUNTREG     EQU 0x10             ;fileReg loc for counter
CNTVAL       EQU d'3'             ;counter value
UNPBCD_ADDR EQU 0x30
ASCII_RESULT EQU 0x40

;-----------main program
        ORG    0
        SETF   TRISB              ;make PORTB input
        CALL   BIN_DEC_CON
        CALL   DEC_ASCII_CON
        BRA    $

;-----converting BIN(HEX) TO DEC (00-FF TO 000-255)
BIN_DEC_CON
        MOVFF PORTB,WREG          ;get the binary data from PORTB
        MOVWF NUME                ;load numerator
        MOVLW MYDEN               ;WREG = 10, the denominator
        CLRF   QU                 ;clear quotient
D_1     INCF   QU                 ;inc quotient for every subtractior
        SUBWF NUME                ;subtract WREG from NUME value
        BC     D_1                ;if positive go back
        ADDWF NUME                ;once too many, first digit
        DECF   QU                 ;once too many for quotient
        MOVFF NUME,RMND_L         ;save the first digit
        MOVFF QU,NUME             ;repeat the process one more time
        CLRF   QU                 ;clear QU
D_2     INCF   QU
        SUBWF NUME                ;subtract WREG from NUME value
        BC     D_2
        ADDWF NUME                ;once too many
        DECF   QU
        MOVFF NUME,RMND_M         ;2nd digit
        MOVFF QU,RMND_H           ;3rd digit
        RETURN

;----converting unpacked BCD digits to displayable ASCII digits
DEC_ASCII_CON
        MOVLW CNTVAL              ;WREG = 10
        MOVWF COUNTREG            ;load the counter, count = 10
        LFSR   0,UNPBCD_ADDR      ;load pointer FSR0
        LFSR   1,ASCII_RESULT     ;load pointer FSR1
B3      MOVF   POSTINC0, W        ;copy RAM to WREG, increment FSR0
        ADDLW 0x30                ;make it an ASCII
        MOVWF POSTINC1            ;copy WREG and increment FSR1
        DECF   COUNTREG,F         ;decrement counter
        BNZ    B3                 ;loop until counter = zero
        RETURN
        END                       ;end of the program
```

232

6.6.5 用存储区作为栈

PIC18的栈有31个字节深，21位宽。正如在第2章讨论过的，程序计数器是21位宽的，因此

栈也是21位宽的。栈主要是用来保存调用和中断子例程的地址。与其他微控制器不同的是，PIC18的栈不是数据RAM的一部分。然而，要访问如此多的存储区，传统的栈显得是没有必要的。在传统的CPU(如x86)中，由于寄存器数量有限，所以不得不在调用子例程的开始就将主要寄存器的内容压入栈保存起来，然后才能使用主要寄存器进行数据操作。就PIC18来讲，在进入子例程时需要从默认的存储区切换到新的存储区。也就是说，在一个给定的子例程中，如果想对程序数据做压栈保护，可以使用数据RAM的一个存储区，来代替31×21位的栈。除了21位宽的栈不能存放8位的数据外，还必须预留31×21位的栈给调用和中断。注意，我们仍然可以使用访问存储区来存储全局变量。全局变量将在下一节讨论。

6.6.6　复习题

1. 把下面的ASCII数字写成ASCII码和压缩BCD码的形式。
   (a)'5','7'　(b)'9','4'
2. 写出"2005"的十六进制和BCD码形式。
3. 下面这条语句执行后，WREG寄存器中还有BCD数据吗?假定MPASM的基数为十进制。
   MOVLW　95
4. 将BCD数33H转换成ASCII码的形式是________H和________H。
5. 请指出11110010B的ASCII值。假定要将它在电脑屏幕上显示成3位的十进制数。
6. 校验和的方法是用来测试________(ROM,RAM)中数据的完整性的。
7. 计算下面一组数据的校验和：88H、99H、AAH、BBH、CCH、DDH。
8. 判断对错：若我们把所有的字节数加起来，包括校验和字节数，结果为FFH，则说明数据没错。

233

6.7　宏和模块

在这一节将探讨宏和模块的概念，以及它们在汇编语言程序设计中的应用。下面将详细地介绍宏的定义和用法，并给出关于宏的诸多应用例子。另外，本节还将介绍模块化编程的概念，以及编写模块程序和程序连接的一些规则。为了说明模块间参数传递的方法，将给出一些实用的模块程序。我们将程序设计成若干模块，使得在其他应用中也能使用这些模块。这样的模块在C语言中被称作函数。通常的做法是，将程序设计成几个模块，分别测试每个模块，然后将它们放到模块库中。

6.7.1　什么是宏以及怎样声明宏

在汇编语言程序设计的许多应用中，常常会见到使用一组指令来完成重复任务的情况。例如，在同一个程序中，反复地向某个RAM地址传送数据。这就没有必要每次都重写代码。所以，为了节省程序编制的时间和减少出错的概率，人们提出了宏的概念。宏允许程序员只写一次任务(为完成特定工作而编写的代码)，在需要用到的地方调用它就可以。

6.7.2　宏的定义

每个宏定义都必须由下面的3部分组成：

名称　　MACRO　　哑元变量1,哑元变量2,……,哑元变量 N

```
……
……
ENDM
```

MACRO伪指令表明宏定义的开始，ENDM伪指令表明宏定义的结束。伪指令MACRO和ENDM之间的部分被称为宏定义体。名称必须是唯一的，并且遵循汇编语言的命名规则。哑元变量可以是名字、参数，或者是宏定义体中用到的寄存器。在宏定义之后，可以使用宏的名字来调用宏定义体，哑元变量将被适当的值取代。将立即数传送到文件寄存器是经常用到的操作，但是没有专门的指令。在这里，可以使用宏定义来完成这项任务，其代码如下：

```
MOVLF MACRO K, MYREG
      MOVLW K
      MOVWF MYREG
      ENDM
```

上面的程序段就是宏定义。注意，哑元变量 K 和 MYREG 在宏定义体中已有提及。

234

下面是关于如何使用上面定义的宏的3个例子。

```
1. MOVLF    0x55, 0x20        ;send value 55H to loc 20H

2. VAL_1    EQU 0x55
   RAM_LOC  EQU 0x20
   MOVLF    VAL_1, RAM_LOC

3. MOVLF    0x55, PORTB       ;send value 55H to Port B
```

指令 MOVLF 0x55,0x20 调用了宏。编译器对宏做扩展，并在列表文件中给出以下的代码：

```
M       MOVLW 5
M       MOVWF 0x20
```

其中M表明这段代码来源于宏。

6.7.3 LOCAL伪指令

到现在为止，所讨论的宏在宏定义体中都没有出现标号或名称。如果宏在程序中被扩展不止一次，并且在宏定义体中的标号域中存在标号，那么必须将这些标号声明成LOCAL。否则在两个或者更多的位置出现同样的标号，编译器将会报错。在宏定义体中存在下面几条规则。

(1) 标号域中的所有标号都必须声明为LOCAL。

(2) LOCAL伪指令必须紧跟在MACRO伪指令之后。也就是说，它必须放在注释和宏定义体之前；否则，编译器将会报错。

(3) LOCAL伪指令一次可以声明多个名称和标号，如：

LOCAL　　名称1,名称2,名称3

或者：

LOCAL　　名称1

LOCAL　　名称2

LOCAL　　名称3

为了澄清上面的几点，请看下面用于时延的宏定义。

```
DELAY_1 MACRO V1, TREG
        LOCAL BACK
        MOVLW V1
        MOVWF TREG
BACK    NOP
        NOP
        NOP
        NOP
        DECF  TREG,F
        BNZ   BACK
        ENDM
```

注意，标号BACK紧跟在MACRO之后且被定义为LOCAL。在其他的任何地方定义标号都会产生错误。LOCAL伪指令允许编译器每次遇到标号时可以分别加以定义。查看列表文件将会发现，当第一次展开宏时，列表文件有"??0000"，在第二次展开宏时有"??0001"，在第三次展开宏时有"??0002"，这都是代替标号BACK的。这表明BACK是局部的。为了区分这些概念，请看程序6-4，观察者没有使用LOCAL时编译器是怎样报错的。下面的代码是使用嵌套循环实现时延的另一个宏定义。

```
DELAY_2 MACRO V1, V2, R1, R2
        LOCAL BACK
        LOCAL AGAIN
        MOVLW V2
        MOVWF R2
AGAIN   MOVLW V1
        MOVWF R1
BACK    NOP
        NOP
        NOP
        NOP
        DECF  R1,F
        BNZ   BACK
        DECF  R2,F
        BNZ   AGAIN
        ENDM
```

现在查看程序6-4，观察在程序中怎样使用宏。

程序6-4 使用宏翻转端口B

```
;----------------------------------------
;Program 6-4: toggling Port B using macros
        #include P18F458.INC

;---------------sending data to fileReg macro
MOVLF MACRO K, MYREG
        MOVLW K
        MOVWF MYREG
        ENDM

;----------------------------time delay macro
DELAY_1 MACRO V1, TREG
        LOCAL BACK
        MOVLW V1
        MOVWF TREG
BACK    NOP
        NOP
```

```
        NOP
        NOP
        DECF   TREG,F
        BNZ    BACK
        ENDM

;----------------------------program starts
        ORG    0
        CLRF      TRISB              ;Port B as an output
OVER    MOVLF     0x55,PORTB
        DELAY_1 0x200,0x10
        MOVLF     0xAA,PORTB
        DELAY_1 0x200,0x10
        BRA       OVER
        END
;--------------------end of file
```


6.7.4 INCLUDE 伪指令

假定有一些宏在每个程序中都要使用到。每次都必须要重写这些宏吗?答案是否定的。使用 INCLUDE 指令就可以解决这个问题。INCLUDE 指令允许程序员编写宏代码并保存到一个文件里,然后允许在任何程序文件中引用它们。例如,假设下面的常用宏指令已编写完毕,并保存为文件名 MYMACRO1.MAC。

假定这些宏都保存在文件 MYMACRO1.MAC 中,使用 INCLUDE 伪指令可以把该宏文件引入到任何".asm"文件中,因此,程序就可以随时调用那些宏。如果文件引用了所有的宏,在对它们进行扩展时,将会在".lst"列表文件的开头列出这些宏,它们是程序的一部分。为了更好地理解这个问题,请参阅程序 6-5。

程序 6-5 使用宏翻转端口 *B*

```
;Program 6-5: toggling Port B using macros
        #include P18F458.INC
        #include "MYMACRO1.MAC" ;get macros from macro file

;---------------------------program starts
        ORG    0
        CLRF   TRISB             ;Port B as an output
OVER    MOVLF  0x55,PORTB
        DELAY_1     0x200,0x10
        MOVLF  0xAA,PORTB
        DELAY_1     0x200,0x10
        BRA    OVER
        END
;--------------------end of file
```

6.7.5 NOEXPAND/EXPAND 伪指令

观察带有宏的.lst 文件,你可以发现宏是被全部展示的。在默认情况下,程序是使用 EXPAND

伪指令的，即宏在每个调用的位置都将显示出来。这样的情况重复两三次是很好的，但是如果重复 237 多次，就会令人讨厌。使用 NOEXPAND 伪指令，我们就可以在列表文件中关闭宏的显示。

```
              00001 ;Program 6-4:toggling Port B using macros
              00002       #include P18F458.INC
              00003       NOEXPAND
              00004 ;---------------sending data to fileReg macro
              00005 MOVLF MACRO K, MYREG
              00006       MOVLW    K
              00007       MOVWF    MYREG
              00008       ENDM
              00009
              00010 ;---------------------------time delay macro
              00011 DELAY_1 MACRO V1, TREG
              00012       LOCAL       BACK
              00013       MOVLW       V1
              00014       MOVWF       TREG
              00015 BACK  NOP
              00016       NOP
              00017       NOP
              00018       NOP
              00019       DECF        TREG,F
              00020       BNZ         BACK
              00021       ENDM
              00022
              00023 ;---------------------------program starts
000000        00024       ORG         0
000000 6A93   00025       CLRF        TRISB       ;Port B as an output
              00026 OVER  MOVLF       0x55,PORTB
              00027       DELAY_1     0x200,0x10
              00028       MOVLF       0xAA,PORTB
              00029       DELAY_1     0x200,0x10
00002A D7EB   00030       BRA         OVER
              00031       END
```

238 图 6-11 程序 6-4 的带有 NOEXPAND 伪指令的列表文件

```
00001 ;Program 6-4:toggling Port B using macros
00002       #include P18F458.INC
00003       EXPAND
00004 ;---------------sending data to fileReg macro
00005 MOVLF MACRO K, MYREG
00006       MOVLW K
00007       MOVWF MYREG
00008       ENDM
00009
00010 ;---------------------------time delay macro
00011 DELAY_1 MACRO V1, TREG
00012       LOCAL BACK
00013       MOVLW V1
00014       MOVWF TREG
00015 BACK  NOP
00016       NOP
00017       NOP
00018       NOP
00019       DECF  TREG,F
```

图 6-12 程序 6-4 的带有 EXPAND 伪指令的列表文件

```
                 00020          BNZ    BACK
                 00021          ENDM
                 00022
                 00023 ;----------------------------program starts
000000           00024          ORG    0
000000 6A93      00025          CLRF   TRISB ;Port B as an output
                 00026 OVER     MOVLF  0x55,PORTB
000002 0E55         M           MOVLW  0x55
000004 6E81         M           MOVWF  PORTB
                 00027 DELAY_1 0x200,0x10
  0000              M           LOCAL  BACK
000006 0E00         M           MOVLW  0x200
000008 6E10         M           MOVWF  0x10
00000A 0000         M  BACK     NOP
00000C 0000         M           NOP
00000E 0000         M           NOP
000010 0000         M           NOP
000012 0610         M           DECF   0x10,F
000014 E1FA         M           BNZ    BACK
                 00028          MOVLF  0xAA,PORTB
000016 0EAA         M           MOVLW  0xAA
000018 6E81         M           MOVWF  PORTB
                 00029 DELAY_1 0x200,0x10
  0000              M           LOCAL  BACK
00001A 0E00         M           MOVLW  0x200
00001C 6E10         M           MOVWF  0x10
00001E 0000         M  BACK     NOP
000020 0000         M           NOP
000022 0000         M           NOP
000024 0000         M           NOP
000026 0610         M           DECF   0x10,F
000028 E1FA         M           BNZ    BACK
00002A D7EB      00030          BRA    OVER
                 00031          END
```


图 6-12 （续）

6.7.6 宏与子例程

宏和子例程在编写汇编程序时都很有用，但两者都有相应的不足。宏在调用时增加了代码的长度。例如，如果一个包括 10 条指令的宏被调用了 10 次，那么程序代码就增加了 100 条指令。然而，如果调用 10 次子例程，代码长度仍保持为子例程指令的长度。子例程的仅有的问题是在调用时要用到栈空间。如果发生了嵌套调用（即在一个子例程中调用另一个子例程），就会产生问题。嵌套调用可能会发生栈溢出，致使程序崩溃。PIC18 对栈溢出的预防，请参阅 PIC18 的参考手册。

6.7.7 模块

在编写软件包时，通常的做法是把整个项目分解成若干个小模块，然后把各个模块分派给不同的程序员完成。这样不仅使整个项目可控性增强，还有一些其他的优点，例如：

(1) 每个模块可以独立地编写、调试和测试；

(2) 一个模块的失败不会让整个项目停止；

(3) 在问题的发现和隔离方面比较容易，并且耗时较少；

(4) 可以将模块和高级语言连接，如C语言；

(5) 并行开发大大缩短了完成项目所需的时间。

下面将介绍怎样编写和连接模块，以创建可执行的程序。

6.7.8 编写模块

在上一节给出的程序中，主程序调用了许多其他的子例程。如果一个子例程不能正常地工作，那么整个程序都要重新编写和编译。开发软件的一个高效的方法是把每个子例程当作具有独立文件名的独立程序(或者模块)，然后分别地编译和测试。在每个程序都测试通过后，就可以整合到一起形成一个完整的程序。为保证各个模块能连接到一起，要使用一些汇编语言伪指令。两个常用的伪指令是 EXTERN(外部) 和 GLOBAL。GLOBAL 伪指令相当于其他编程语言中的 PUBLIC 指令。下面分别加以讨论。

240

6.7.9 EXTERN 伪指令

EXTERN 伪指令用来告诉编译器和链接器：一些名字和变量不是在当前模块中定义的，而是在外部某个地方定义的。如果不使用 EXTERN 伪指令，那么编译器就会报错，因为它找不到定义名称的位置。EXTERN 伪指令的格式如下：

EXTERN 名称 1;每个名称的声明可以使用单独的 EXTERN 伪指令

EXTERN 名称 2

EXTERN 名称 1,名称 2;或者在同一个 EXTERN 伪指令中列出

6.7.10 GLOBAL 伪指令

如果名称或参数已被定义为 EXTERN(表明它们在当前模块外定义)，那么它们在所属模块中必须定义为 GLOBAL。将名称定义为 GLOBAL(PUBLIC) 允许编译器和连接器将它与 EXTERN 定义相匹配。GLOBAL 伪指令的格式如下：

GLOBAL 名称 1;每个名称的声明可以使用单独的 GLOBAL 伪指令

GLOBAL 名称 2

GLOBAL 名称 1, 名称 2;或者在同一个 GLOBAL 伪指令中列出

程序 6-6 可以帮助你理清这些概念。它表明，对于每个 EXTERN 定义，必然在其他的模块中有一个 GLOBAL 与之相对应。请注意程序的入口和出口。被主模块调用的子模块都有自己的 END 伪指令。请看例 6-6。

程序 6-6

```
;------------------------------------------------
;PROG 6-6: MAIN.ASM - CALCULATING AND TESTING CHECKSUM BYTE
        #include P18F458.INC

RAM_ADDR        EQU 40H
COUNTREG        EQU 0x20            ;fileReg loc for counter
CNTVAL          EQU 4               ;counter value
CNTVAL1         EQU 5               ;counter value

        EXTERN CAL_CHKSUM
        EXTERN TEST_CHKSUM

PGM CODE
;------------main program
        ORG     0
        CALL    COPY_DATA           ;this subroutine is in this file
        CALL    CAL_CHKSUM          ;this sub is in external file
        CALL    TEST_CHKSUM         ;this sub is in external file
        BRA     $
;--------copying data from code ROM to data RAM
COPY_DATA
        MOVLW low(MYBYTE)           ;WREG = 00 LOW-byte addr.
        MOVWF TBLPTRL               ;ROM data LOW-byte addr.
        MOVLW hi(MYBYTE)            ;WREG = 5, HIGH-byte addr.
        MOVWF TBLPTRH               ;ROM data HIGH-byte addr.
        MOVLW upper(MYBYTE)         ;WREG = 00 upper-byte addr.
        MOVWF TBLPRTRU              ;ROM data upper-byte addr.
        LFSR  0,RAM_ADDR            ;FSR0 = RAM_ADDR, place to save
C1      TBLRD*+                     ;bring in next byte and inc TBLPTR
        MOVF  TABLAT,W              ;copy to WREG (Z = 1, if null)
        BZ    EXIT                  ;is it null char? exit if yes
        MOVWF POSTINC0              ;copy WREG to RAM and inc pointer
        BRA   C1
EXIT  RETURN
;----------my data in program ROM
      ORG 0x500
MYBYTE DB 0x25, 0x62, 0x3F, 0x52, 0x00
      END

;------------------------------------------------
;PROG 6-6: CALCCSB.ASM - CALCULATING CHECKSUM BYTE
      #include P18F458.inc

RAM_ADDR        EQU 40H
COUNTREG        EQU 0x20            ;fileReg loc for counter
CNTVAL          EQU 4               ;counter value
CNTVAL1         EQU 5               ;counter value

      GLOBAL CAL_CHKSUM
```

241

```
PGM CODE                      ;we use this to inform the linker that
                              ;the code segment has the name PGM
CAL_CHKSUM
      MOVLW CNTVAL            ;WREG = 4
      MOVWF COUNTREG          ;load the counter
      LFSR  0,RAM_ADDR        ;load pointer. FSR0 = 40H
      CLRF  WREG
C2    ADDWF POSTINC0,W        ;add RAM to WREG and increment FSR0
      DECF  COUNTREG,F        ;decrement counter
      BNZ   C2                ;loop until counter = zero
      XORLW 0xFF              ;1's comp
      ADDLW 1                 ;2'compl
      MOVWF POSTINC0
      RETURN
      END

;----------------------------------------------------------
;PROG 6-6: TESTCSB.ASM - TESTING CHECKSUM BYTE
      #include P18F458.inc

RAM_ADDR     EQU 40H
COUNTREG     EQU 0x20         ;fileReg loc for counter
CNTVAL       EQU 4            ;counter value
CNTVAL1      EQU 5            ;counter value
      GLOBAL TEST_CHKSUM
      PGM CODE
TEST_CHKSUM
      MOVLW CNTVAL1           ;WREG = 5
      MOVWF COUNTREG          ;load the counter
      CLRF  TRISB
      LFSR  0,0x40            ;load pointer. FSR0 = 40H
      CLRF  WREG
C3    ADDWF POSTINC0,W        ;add RAM and increment FSR0
      DECF  COUNTREG,F        ;decrement counter
      BNZ   C3                ;loop until counter = zero
      XORLW 0x0               ;EX-OR to see if zero
      BZ    G_1               ;is result zero? then good
      MOVLW 'B'
      MOVWF PORTB             ;if not, data is bad
      RETURN
G_1   MOVLW 'G'
      MOVWF PORTB             ;data is not corrupted
      RETURN
      END
```

242

6.7.11 连接模块

假定程序6-6的每个程序模块都已分别编译完毕，且分别保存为文件MAIN.O、CALCCSB.O和TESTCSB.O，下面的命令将演示怎样使用MPLINK来将它们连接在一起，生成一个可执行文件：

>MPLink.exe"18f458.1kr""MAIN.O""CALCCSB.O""TESTCSB.O"/o"PRG6-.COF"

程序6-6还表明伪指令EXTERN和GLOBAL也可用于数据变量。连接器通过匹配

GLOBAL 和 EXTERN 定义的名称，解决了外部引用的问题。连接器程序将在 MPLINK 命令指定的文件范围内搜索外部子例程。

MPLAB IDE 能一步到位地处理编译和连接。这就有助于降低程序开发时间并避免在输入 MPLINK 命令行时发生错误。

6.7.12 复习题

1. 讨论宏指令编程的优点。
2. 请列出宏指令的 3 个组成部分。
3. 解释和比较宏定义、调用宏和扩展宏的概念。
4. 判断对错：在宏内定义的标号可以被编译器自动地识别为局部变量(LOCAL)。
5. 在模块内使用伪指令________，表明所命名的变量或子例程可以用在其他的模块中。
6. 在模块内使用伪指令________，表明所命名的变量或子例程已在其他模块中定义。

小结

本章介绍了 PIC18 的寻址方式。立即寻址方式使用常数作为操作数。直接和寄存器间接寻址方式可用来访问存储在 PIC18 数据 RAM 文件寄存器中的数据。寄存器间接寻址方式使用一个指针来指向数据，这种寻址方式的优点是可以动态地寻址。变址 ROM 寻址方式常用来访问 PIC18 的程序 ROM 中的查询表数据。PIC18 允许读取程序 ROM 里的定值数据，除此之外，还允许向带有 flash ROM 的 PIC18 写入 ROM 数据。

被称作 SFR 的一组寄存器能够通过它们的名称或地址来访问。本章也讨论了具备位寻址能力的数据 RAM 和端口，以及怎样用位操作指令来直接地访问它们。同时，本章还讨论了存储区转换的概念，并介绍了怎样使用 BSR 寄存器来访问 PIC18 的 16 个存储区。

本章还介绍了如何将一个程序分解成若干个子例程，并分别编写与测试。最后介绍了宏和模块的概念以及它们的优点。

习题

1. 在下面立即寻址方式的指令中，哪些是非法的？
   (a)MOVLW 0x24 (b) MOVLW MYREG,0x30 (c)MOVLW 0x60
2. 区别下列指令的寻址方式：
   (a)MOVWF PORTB (b)MOVLW 0x50 (c)MOVWF MYREG
   (d)MOVLW 0 (e)MOVFF MYBREG, YOUREG
   (f) MOVWF YOURREG
3. 指出分配给下列寄存器的地址：
   (a) PORTB (b) WREG (c) PORTC (d) PORTD (e) PCL (f) PCH
   (j) PCU (h) TRISC (i) TRISB (j) STATUS (k) FSR0L
4. 哪一个存储区是用于 SFR 的？
5. 在访问 SFR 时，必须使用________寻址方式。
6. 指出下面这条指令的作用：MOVLW 0xF0。

7. 指出下面这条指令的作用:MOVWF PORTC。
8. 指出下面这条指令的作用:MOVF PORTC,W。
9. 指令 CLRF MYREG 是________(合法的,非法的)。
10. 分配给数据 RAM 的低 128 B 的地址范围是________到________。
11. SFR 的地址范围是________到________。
12. 请指出分配给下面两组寄存器的地址范围?它们之间有逻辑间隔吗?

244

    (a) 访问存储区的 RAM 空间　　(b) 访问存储区的 SFR
13. 编制程序,将 6、9、2、5、7 相加,并把结果保存到 RAM 地址 20H。
14. 当访问数据 RAM 时,哪些寄存器可用作寄存器间接寻址方式的指针?请给出它们的名称以及赋值方法。
15. 编制程序,复制数据 55H 到 RAM 地址 50H～60H。
16. 编制程序,将起始地址为 40H 的 RAM 中的 10 B 数据复制到起始地址为 70H 的 RAM 中。
17. FSRx 寄存器的大小是________。
18. 请给出同 FSR0 和 FSR1 有关的 SFR 寄存器。
19. 编制程序,将 RAM 地址 0H～7FH 的内容清空。
20. 编制程序,将 RAM 地址 50H～5FH 的内容取反。
21. 请解释 INFx 寄存器的作用。
22. FSRx 寄存器能覆盖多大的 RAM 空间?
23. 编译程序,对于下面的数据,请说明每个 ROM 地址的内容。

```
            ORG     200H
MYDAT_1:    DB      "Earth"
MYDAT_2:    DB      "987-65"
MYDAT_3:    DB      "GABEH 98"
```

24. 编译程序,对于下面的数据,请说明每个 ROM 地址的内容。

```
            ORG    340H
 DAT_1: DB 0x22,0x56, B'10011001', D'32', 0xF6, B'11111011'
```

25. 当要访问的数据存储在程序 ROM 中时,哪些寄存器可用作寄存器间接寻址方式的指针?给出它们的名称以及赋值方法。
26. 请解释寄存器 TABLAT 的作用。
27. 请指出 TBLPTR 寄存器的大小,以及它可以覆盖的程序 ROM 空间。
28. 请给出同 TBLPTR 有关的 SFR。
29. 编制程序,从 ROM 中读取下面的数据,并放到起始地址为 50H 的 RAM 中。

```
            ORG  0x600
MYDATA      DB     "1-800-999-9999",0
```

30. 编制程序,求 $y=x^2+2x+5$ 中 y 的值,其中 x 的取值范围为 0～9。
31. 编制程序,求 $y=20x+5$ 中 y 的值,其中 x 的取值范围为 0～9。
32. 编制程序,从 ROM 中读取下面的数据,并放到起始地址为 40H 的 RAM 中。

```
             ORG  0x700
MYDATA       DB     "The earth is but one country",0
```

33. 判断对错:读表指令适用于 PIC18 系列所有型号的芯片。

245

34. 判断对错:写表指令适用于 PIC18 系列带有 flash ROM 的芯片。
35. 假定端口 B 的低 4 位与 4 个开关相连。编制程序,根据开关的状态,发送对应的 ASCII 字符到端口 C。

0000　　'0'
0001　　'1'

0010	'2'
0011	'3'
0100	'4'
0101	'5'
0110	'6'
0111	'7'
1000	'8'
1001	'9'
1010	'A'
1011	'B'
1100	'C'
1101	'D'
1110	'E'
1111	'F'

36. 编制程序，在 RB5 位上产生占空比为 75% 的方波。
37. 编制程序，在 RC7 位上产生占空比为 80% 的方波。
38. 编制程序，监视 RB4 位的状态。当它变为高电平时，程序在 RB7 位上发出声音（占空比为50% 的方波）。
39. 编制程序，监视 RC1 位的状态。当它变为低电平时，程序发送数据 55H 到 RD0 位。
40. 进位标志位属于哪个寄存器？
41. 零标志位的位地址是多少？
42. 下面哪些指令是合法的？如果是合法的，指出哪一位将被改变。
   (a) BSF PORTB,1　(b) BSF PORTC,3　(c) BCF WREG,1
   (d) BCF 0x30,1　(e) BCF PORTD,0　(f) BST STATUS,C
   (g) CLRF WREG,3　(h) CLRF FSR0
43. 指令 BTG PORTB,0 是________（合法的，非法的）。
44. PORTB、PORTC 和 PORTD 中哪些 I/O 端口是可以位寻址的？
45. PIC18 的哪些寄存器可以位寻址？
46. 请给出一条清空进位标志位的指令。
47. 请说说怎样检测进位标志位是否为高电平。
48. 请说说怎样检测零标志位是否为高电平。
49. 请给出状态寄存器的 C、Z、DC 和 OV 的位地址。
50. 判断对错：位地址 0 ～ 7 被分配给了 000 ～ FFFH 范围内的每一个 RAM 地址。
51. 判断对错：SFR 不能够位寻址。
52. 编写指令，保存 C 标志位到内存 10H 的第 4 位。
53. 编写指令，保存 DC 标志位到内存 16H 的第 2 位。
54. 编写指令，保存 Z 标志位到内存 12H 的第 7 位。
55. 编写指令，检查 WREG 寄存器的 D0 位和 D1 位是否为高电平。如果是，那么将 WREG 的值除以 4。
56. 编制程序，检查 WREG 寄存器的 D7 位是否为高电平。如果是，那么发送信息到 LCD，以说明 WREG 的内容为负数。
57. 编制程序，将 RAM 地址 20H 的所有位都置 1，要求使用下面的两种方法：
   (1) 字节寻址　　(2) 位寻址

246

58. 编制程序,判断 WREG 的内容是否能被 8 整除。
59. 编制程序,找出文件寄存器地址 05H 中零的个数。
60. 访问 SFR 要使用哪种寻址方式?
61. 哪种寻址方式是用来访问 PIC18 最后 128 B RAM 的?
62. 请给出访问存储区低 128 B 和高 128 B 的地址范围。
63. 在 PIC18 中,SFR 都采用相同的地址,其地址范围是________到________。
64. PIC18 最多有________个存储区。
65. 分析下面两条指令的不同之处:
    (a) ADDWR MYREG,F,1 (b) ADDWR MYREG,F,0
66. 哪种寻址方式可用来访问不同的存储区?
67. 编制程序,将数据 55H 传送到 RAM 地址 1C0H ~ 1CFH。
68. 编制程序,将 RAM 地址 20H ~ 2FH 的内容复制到 RAM 地址 2D0 ~ 2DFH。
69. 分析下面两条指令的不同之处:
    (a) CLRF MYREG,F,1 (b) CLRF MYREG,F,0
70. 分析下面两条指令的不同之处:
    (a) SETF MYREG,W,1 (b) SETF MYREG,W,0
71. 分析下面两条指令的不同之处:
    (a) INCF MYREG,F,1 (b) INCF MYREG,F,0
72. 计算 ASCII 数据 Hello 的校验和字节数。
73. 判断对错:若将所有字节数相加,包括校验和字节数,所得结果为 00H,则说明数据无误。
74. 编制程序:(a) 从程序 ROM 中读取数据"Hello, my fellow world citizens",(b) 计算校验和字节数,(c) 验证校验和。

247

75. 要在 LCD 或电脑屏幕上显示数据,则它必须为________格式(二进制、BCD、ASCII)。
76. 假定端口 B 的低 4 位分别与 4 个开关相连。编制程序,根据开关的状态,发送对应的 ASCII 字符到端口 D。

0000	'0'
0001	'1'
0010	'2'
0011	'3'
0100	'4'
0101	'5'
0110	'6'
0111	'7'
1000	'8'
1001	'9'

77. 编制程序,将压缩 BCD 码转换为 ASCII 码。假定压缩 BCD 码位于起始地址为 700H 的 ROM 中。将 ASCII 码放到其实地址为 40H 的 RAM 中。

```
            ORG  700H
MYDATA      DB   76H,87H,98H,43H
```

78. 编制程序,将 ASCII 码转换为压缩 BCD 码。假定 ASCII 码位于起始地址为 300H 的 ROM 中。将 BCD 码放到起始地址为 60H 的 RAM 中。

```
            ORG  300H
MYDATA      DB   "87675649"
```

79. 编制程序，从 PD 读入 8 位二进制数据，转换成 ASCII 码，然后将结果保存到 RAM 地址 40H、41H 和 42H 中。如果 PD 的输入是 1000 1101，那么结果是什么呢？
80. 请说出宏的两个优点。
81. 哪个使用了更多的程序 ROM 空间：宏还是模块？
82. 为什么编写程序时要用到模块？请给出 3 点理由。
83. 如果一个标号或参数没有在给定的模块中定义，那么它必须声明为________。
84. 如果一个标号或参数将被其他的模块使用，那么它必须声明为________。
85. 使用宏和模块重做第 79 题。

248

复习题答案

6.1 节

1. 否　　2. MOVLW B'10000000'
3. PIC 不允许将常数直接传送给文件寄存器。 4. 正确。 5. 错误。

6.2 节

1. 直接寻址方式。存储地址是 0x40。
2. 数据 RAM 文件寄存器的 12 位地址的低 8 位。地址为 0FE9H。
3. 数据 RAM 文件寄存器的 12 位地址的高 4 位。地址为 0FEAH。
4. 12 位　　5. FSR0、FSR1、FSR2。

6.3 节

1. TBLPTR　　2. TBLPTR　　3. TABLAT　　4. 21 位，2 MB　　5. TBLPTR
6. 在 TBLRD*＋中，先读取表单内容，然后 TBLPTR 加 1。
   在 TBLRD+* 中，TBLPTR 先加 1，然后读取表单内容。
7. 错误。只适用于带有 flash ROM 的芯片。

6.4 节

1. 正确。 2. 正确。 3. 正确。 4. a、b、c 和 d。 5. 全部。 6. BTFSS 0x03, 1
7. (a) 将 RAM 地址 20H 的第 1 位置 1。
   (b) 将 RAM 地址 32H 的第 7 位清零。
   (c) 将 RAM 地址 12H 的第 2 位置 1。
   (d) 将端口 B 的第 4 位置 1。
   (e) 将状态寄存器的第 1 位置 1。
8. BCF STATUS, C

6.5 节

1. 正确。 2. 正确。 3. 正确。 4. F80～FFFH

249

5.
```
MYREG EQU    0x2
MOVLB 0x02        ;load 2 into BSR (use bank 2)
MOVLW 0x99        ;WREG = 99h
MOVWF MYREG,1
```

6.
```
MYREG EQU   0x08
MOVLB 0x4         ;load 4 into BSR (use bank 4)
MOVLW 0x55        ;WREG=55h
MOVWF MYREG,1
```

7. 正确。 8. 8位,4

6.6 节

1. 由35H和37H合成的BCD码是57H,由39H和34H合成的BCD码为94H。
2. ASCII数据是32H、30H、30H、35H,而05H和20H是BCD码。
3. 否。要转换成BCD码,则基数必须设为十六进制。
4. 33H和33H 5. 242或32H,34H和32H 6. ROM
7. 88H+99H+AAH+CCH+DDH=42FH。去掉进位,得到2FH,求补后得到D1H。
8. 错误。

6.7 节

1. 宏指令编程允许在程序中使用一条语句来调用频繁使用的一组指令,从而节省程序员的时间,并且使得程序可读性更好。
2. 宏的3个组成部分是MACRO伪指令、宏定义体和ENDM伪指令。
3. 宏定义是一系列将要执行的宏的声明。宏定义以MACRO开始,以ENDM结束。宏可以在汇编语言程序的任何地方调用。当汇编程序使用宏定义体中的汇编语言替换调用宏的命令行时,就是宏的扩展。
4. 错误。宏的局部的标号必须使用LOCAL伪指令声明为局部变量。
5. GLOBAL伪指令 6. EXTERN伪指令

第 7 章 PIC C 语言编程

学习目标：

- ☐ PIC18 的 C 数据类型
- ☐ 时延与 I/O 操作的 C18 编程
- ☐ I/O 位操作的 C18 编程
- ☐ 逻辑操作和算术操作的 C18 编程
- ☐ ASCII 码和 BCD 码转换的 C18 编程
- ☐ 二进制(十六进制)与十进制转换的 C18 编程
- ☐ 数据串行化的 C18 编程
- ☐ C18 C 编译器的 RAM 和 ROM 分配

编译器生成可供下载到微控制器 ROM 中的十六进制文件，其大小是程序员关心的主要问题之一，其原因有 2 个：

(1) 微控制器只有有限的片上 ROM；

(2) PIC18 的程序存储空间的上限只有 2 MB。

编程语言的选择是如何影响编译程序的大小的呢？汇编语言生成的十六进制文件比 C 语言生成的十六进制文件要小得多，但编写汇编程序是繁琐而耗时的事情。另一方面，C 语言编程不那么耗时，而且容易编写，但 C 编译器生成的十六进制文件较汇编器生成的十六进制文件就大得多。下面是选择 C 语言编程的几点主要理由：

(1) 用 C 语言编程比用汇编语言更容易、更省时；

(2) 使用 C 语言编程更容易修改和更新；

(3) 可以使用函数库中的代码；

(4) C 语言代码可以方便地移植到其他的微控制器上。

一些第三方公司开发了 PIC 微控制器的 C 编译器。这里的目标不是逐个地介绍，而是介绍 PIC18 C 编程的基本知识。对于本章的例子和程序，读者可以自己选择编译器。本书采用的是 Microchip 公司的 C18 C 编译器，并同 MPLAB IDE 相结合。Microchip 公司在其网站上提供 C18 C 编译器的学生版下载。请访问 http://www.MicroDigtalED.com，获取 C18 C 编译器和 MPLAB 仿真器的使用指南。

本章的主题是 PIC18 的 C 语言编程。7.1 节将讨论数据类型和时延。7.2 节将介绍 I/O 端口编程。在 7.3 节将讨论逻辑操作 AND、OR、XOR、取反和移位操作。7.4 节将讨论 ASCII 码和 BCD 码之间的转换以及校验和。7.5 节将介绍数据的串行化。7.6 节将介绍 C18 C 编译器如何使用程序 ROM 来存储数据。7.7 节将介绍 C18 的数据 RAM 分配。

7.1 C语言中的数据类型和时延

本节将首先讨论C语言的数据类型，然后给出用于时延的程序代码。

7.1.1 PIC18的C语言数据类型

C18程序员的目标之一就是生成更小的十六进制文件，因此有必要重新回顾C18的C语言数据类型。也就是说，对C18的C语言数据类型的正确理解有助于程序员生成较小的十六进制文件。在这一节中，将集中讨论一些特定的C语言数据类型，它们在PIC18微控制器上是非常有用的，并且广为使用。表7-1列出了各种数据类型及其大小范围。

252

表7-1 C18常用的数据类型

数据类型	位　数	数据范围/使用方法
无符号字符	(8位)	0～255
字符	(8位)	−128～+127
无符号整型	(16位)	0～65 535
整型	(16位)	−32 768～32 768
无符号短整型	(16位)	0～65 535
短整型	(16位)	−32 768～32 768
无符号超短长整型	(24位)	0～16 777 215
超短长整型	(24位)	−8 388 608～+8 388 607
无符号长整型	(32位)	0～4 294 967 295
长整型	(32位)	−2 147 483 648～+2 147 483 648

7.1.2 无符号字符

因为PIC18是一种8位的微控制器，所以字符数据类型是多数应用的首选。无符号字符型(unsigned char)是8位的数据类型，取值范围为0～255(00～FFH)。这是PIC18最常用的数据类型之一。在很多情况下，如设置计数值，不需要有符号数据，此时应该用无符号字符，而不用有符号字符(signed char)。记住：C编译器默认使用带符号字符，除非在字符前加关键字*unsigned*(如例7-1所示)。无符号字符数据类型可以用来表示ASCII字符串，包括扩展ASCII字符串。例7-2展示了ASCII字符串的使用。例7-3说明了端口的翻转。

在声明变量时，必须注意数据的大小，尽量使用无符号字符型代替整型。因为PIC18微控制器只有有限数量的寄存器和数据RAM空间，使用整型会导致更大的十六进制文件。数据类型的误用在其他编译器上可能不是很严重的问题，如x86 IBM PC上的Microsoft Visual C++。

例7-1 编制C18程序，发送数据00～FF到端口*B*。

解：

```
#include <P18F458.h>              //for TRISB and PORTB declarations
void main(void)
  {
    unsigned char z;
```

```
  TRISB = 0;              //make Port B an output
  for(z=0;z<=255;z++)
    PORTB = z;
  while(1);               //NEEDED IF RUNNING IN HARDWARE
}
```

在仿真器上运行上面的程序,查看端口 *B* 是如何显示二进制数 00~FFH 的。注意,若程序在硬件上运行,则 while(1)是必需的。

253

例 7-2 编写 C18 程序,发送字符 0、1、2、3、4、5、A、B、C 和 D 的 ASCII 码到端口 *B*。

解:

```
#include <P18F458.h>
void main(void)
  {
   unsigned char mynum[]= "012345ABCD";//data is stored in RAM
   unsigned char z;
   TRISB = 0;              //make Port B an output
   for(z=0;z<10;z++)
     PORTB = mynum[z];
   while(1);               //stay here forever
  }
```

在仿真器上运行上面的程序,查看端口 *B* 是如何显示 30H、31H、32H、33H、34H、35H、41H、42H、43H 和 44H(即 0、1、2 等的 ASCII 码)的。注意,如果程序在硬件上运行,while(1)是必需的,这很像汇编程序里的指令 GOTO $ 或者 BRA $ 。

例 7-3 编写 C18 程序,连续地反转端口 *B* 的所有位。

解:

```
// Toggle PB forever
#include <P18F458.h>
void main(void)
  {
   TRISB = 0;          //make Port B an output
   for(;;)             //repeat forever
     {
      PORTB = 0x55;    //0x indicates the data is in hex (binary)
      PORTB = 0xAA;
     }
}
```

在仿真器上运行上面的程序,查看端口 *B* 是如何连续地翻转的。

Special Function Registers

Address	SFR Name	Hex	Decimal	Binary	Char
0F76	TXERRCNT	00	0	00000000	.
0F80	PORTA	00	0	00000000	.
0F81	PORTB				
0F82	PORTC	00	0	00000000	.
0F83	PORTD	00	0	00000000	.

254

图 7-1 使用 MPLAB 查看 SFR 的值

7.1.3 有符号字符

有符号字符(signed char)是8位的数据类型,最高有效位(即D7~D0的D7位)表示正数或负数。其余的7位表示有符号数的大小,即有符号数的取值范围是-128~+127。当需要用正数、负数来表示数量(如温度)的时候,有符号字符数据类型就很有用。

再次强调,如果不使用关键字 *unsigned*,默认的数据类型将是有符号数。应该坚持使用无符号字符,除非数据一定要用有符号数表示。

例7-4 编写C18程序,发送数据-4~+4到端口*B*。

解:

```
//sign numbers
#include <P18F458.h>
void main(void)
  {
    char mynum[]= {+1,-1,+2,-2,+3,-3,+4,-4};
    unsigned char z;
    TRISB = 0;                //make Port B an output
    for(z=0;z<8;z++)
      PORTB = mynum[z];
    while(1);                 //stay here forever
  }
```

在仿真器上运行上面的程序,查看端口*B*是如何显示1、FFH、2、FEH、3、FDH、4和FCH(即+1、-1、+2、-2等的十六进制形式)的。关于有符号数的讨论,请参阅第5章。

7.1.4 无符号整型

无符号整型(unsigned int)是16位的数据类型,取值范围为0~65 535(0000~FFFFH)。在PIC18中,无符号整型是用来定义16位变量的,如存储地址。它也可以用来设置计数值大于256的计数器。因为PIC18是8位的微控制器,并且整型数据需占2 B的RAM空间,所以在万不得已的情况下才使用整型数据类型。考虑到寄存器和存储器都在8位的块中,所以误用整型变量会导致十六进制文件更大。这种误用对于配置为512 MB内存、32位奔腾寄存器和存储器、133 MHz总线速度的PC来说并不是什么问题,然而,对于PIC18编程来说,在能用无符号字符的地方就不要使用有符号字符。当然,对于这种误用,编译器不会报错,但是对十六进制文件大小的影响是相当显著的。同样,在不需要使用有符号数据时(如设置计数值),应该使用无符号整型代替有符号整型,这样数据声明范围更广。再次强调:C编译器默认使用有符号整型,除非使用关键字 *unsigned* 说明。

255

7.1.5 有符号整型

有符号整型是16位的数据类型,用最高有效位(即D15~D0的D15位)来表示正负。因此,只有15位可用来表示数值的大小,取值范围是-32768~32767。

7.1.6 其他数据类型

无符号整型的取值范围是0~65535(0000~FFFFH)。C18 C编译器还支持超短长整型和

长整型两种数据类型，可用于声明多于 16 位的变量。请参阅表 7-1。超短长整型是 24 位的，而长整型是 32 位的。

例 7-5 编写 C18 程序，将端口 *B* 的所有位翻转 50 000 次。

解：

```
#include <P18F458.h>
void main(void)
  {
     unsigned int z;
     TRISB = 0;                //make Port B an output
     for(z=0;z<=50000;z++)
       {
         PORTB = 0x55;
         PORTB = 0xAA;
       }
    while(1);                  //stay here forever
  }
```

在仿真器上运行上面的程序，查看端口 *B* 是怎样连续地翻转。注意无符号长整型的最大上限值是 65 535。

例 7-6 编写 C18 程序，将端口 *B* 的所有位翻转 100 000 次。

解：

```
//toggle PB 100,00 times
#include <P18F458.h>
void main(void)
  {
     unsigned short long z;
     unsigned int x;
     TRISB = 0;                //make Port B an output
     for(z=0;z<=100000;z++)
       {
         PORTB = 0x55;
         PORTB = 0xAA;
       }
     while(1);                 //stay here forever
  }
```

256

7.1.7 时延

在 C18 中有两种方法可用来创建时延：

(1) 使用简单的 for 循环；

(2) 使用 PIC18 定时器。

在用这两种方法编写时延程序时，必须用示波器测量具体的延迟时间。接下来将介绍使用 for 循环创建时延。使用 PIC18 定时器创建时延的内容将在第 9 章中介绍。

在用 for 循环创建时延时，必须考虑下面的两个因素，它们直接影响到延迟时间的精度。

(1) 晶振频率是影响时延计算的最主要因素。晶振连接到 OSC1～OSC2 输入引脚之间。

指令周期(即时钟周期的持续时间)是晶振频率的函数。

(2) 影响时延的第二个因素是用来编译 C 程序的编译器。当用汇编语言写程序时,可以控制具体的指令和它们在时延子例程中的顺序。在 C 语言程序中,将 C 语句和函数转换为汇编指令的是 C 编译器。因此,不同的编译器将生成不同的代码。也就是说,如果在不同的编译器上编译相同的 C 程序,会生成不同大小的十六进制文件。

基于以上两点理由,当用 C 语言编写时延程序时,必须使用示波器观察精确的时延时间。请看例 7-7 和例 7-8。

例 7-7 编写 C18 程序,连续地翻转端口 *B* 的所有位,时间间隔为 250 ms。假定系统是 PIC18F485,XTAL=10 MHZ。

解:

```
#include <P18F458.h>
void MSDelay(unsigned int);
void main(void)
  {
    TRISB = 0;                //make Port B an output
    while(1)                  //repeat forever
      {
        PORTB = 0x55;
        MSDelay(250);
        PORTB = 0xAA;
        MSDelay(250);
      }
  }

void MSDelay(unsigned int itime)
  {
    unsigned int i; unsigned char j;
    for(i=0;i<itime;i++)
      for(j=0;j<165;j++);
  }
```


例 7-8 编写 C18 程序,连续地翻转端口 *C* 和端口 *D* 的所有位,时间间隔为 250 ms。

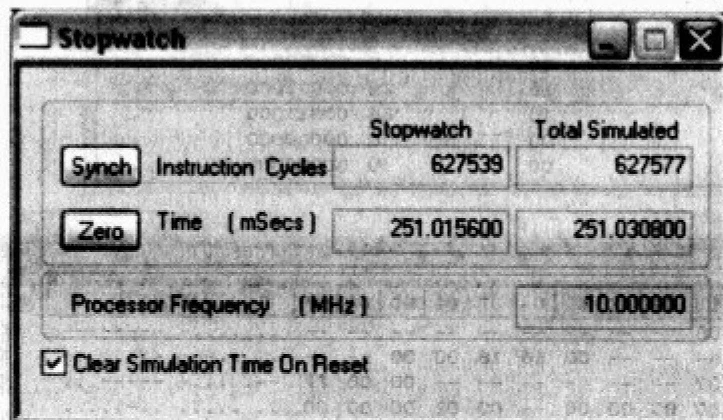
解:

```
//this program is tested for the PIC18F458 with XTAL = 10 MHz
#include <P18F458.h>
void MSDelay(unsigned int);
void main(void)
  {
    TRISC = 0;
    TRISD = 0;                //make Ports C and D output
    while(1)                  //another way to do it forever
      {
        PORTC = 0x55;
        PORTD = 0x55;
        MSDelay(250);
        PORTC = 0xAA;
        PORTD = 0xAA;
        MSDelay(250);
      }
```

```
  }
void MSDelay(unsigned int itime)
  {
    unsigned int i; unsigned char j;
    for(i=0;i<itime;i++)
      for(j=0;j<165;j++);
  }
```

7.1.8 复习题

1. 请给出无符号字符型和有符号字符型数据的取值范围。
2. 请给出无符号整型和有符号整型数据的取值范围。
3. 如果要声明一个用于个人年龄的变量,那么应该使用________数据类型。
4. 判断对错:如果希望程序代码能移植到其他 PIC18 系统上,那么不推荐使用 for 循环来创建时延。
5. 请给出影响时延大小的两个因素。



MPLAB 中的秒表功能允许程序员在编写微控制器程序之前查看时延

图 7-2 用 MPLAB 测量例 7-8 的时延

258

7.2 C 语言 I/O 编程

在这一节中,将介绍 PIC18 的 I/O 端口的 C 语言编程,包括 I/O 端口的字节编程和位编程。

7.2.1 字节 I/O 编程

正如在第 4 章提到的,端口 PORTA~PORTD 都可以按字节访问。在 C18 的头文件中,PORTA~PORTD 的标号已有定义,程序员可以直接使用,如例 7-9 所示。研究下面的几个例子,可以更好地理解 C18 的端口访问。

例 7-9 在端口 B 和端口 C 的引脚上连接有 LED。编制 C18 程序,在 LED 上显示数据 0~FFH(二进制数为 0000 0000~1111 1111B)。

解:

```
#include <P18F458.h>
#define LED PORTC                  //notice how we can define Port C
void main(void)
  {
     TRISB = 0;                    //make Port B an output
     TRISC = 0;                    //make Port C an output
     PORTB = 00;                   //clear Port B
     LED = 0;                      //clear Port C
     for(;;)                       //repeat forever
       {
          PORTB++;                 //increment Port B
          LED++;                   //increment Port C
       }
  }
```

Special Function Registers

Address	SFR Name	Hex	Decimal	Binary	Char
0F80	PORTA	00	0	00000000	.
0F81	PORTB	18	24	00011000	.
0F82	PORTC	18	24	00011000	.
0F83	PORTD	00	0	00000000	.
0F84	PORTE	00	0	00000000	.

File Registers

Address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
0F70	00	00	00	00	00	00	00	--	--	--	--	--	--	--	--	--	.......- --------
0F80	00			00	00	--	--	--	--	00	18	18	00	00	--	--	.....--- -.....--
0F90	--	--	7F	00	00	FF	07	--	--	--	--	--	--	00	00	FF	--.....- -----...
0FA0	00	00	5F	00	00	FF	00	00	00	00	--	00	02	00	00	00	.._..... ..-.....
0FB0	00	00	00	00	00	00	00	00	--	--	00	00	00	00	00	00	.... --......
0FC0	--	00	00	00	00	00	00	00	00	00	00	FF	00	00	00	00	-....... ....
0FD0	1C	00	05	00	--	FF	00	00	00	00	05	--	--	--	--	--	....-... ...-----

Hex　Symbolic

图 7-3　例 7-9 运行 24 次循环后的结果

例 7-10　编写 C18 程序,从端口 *B* 读取 1 B 的数据。在等待 0.5 s 后,将其发送到端口 *C*。

解:

```
#include <P18F458.h>
void MSDelay(unsigned int);
void main(void)
  {
     unsigned char mybyte;
     TRISB = 0xFF;                 //Port B as input
     TRISC = 0;                    //Port C as output
     while(1)
       {
```

```
            mybyte = PORTB;        //get a byte from Port B
            MSDelay(500);
            PORTC = mybyte;        //send it to Port C
          }
    }

void MSDelay(unsigned int itime)
    {
      unsigned int i;
      unsigned char j;
      for(i=0;i<itime;i++)
        for(j=0;j<165;j++);
    }
```

例 7-11 编写 C18 程序，从端口 *C* 读取 1B 的数据。若小于 100，则将其发送到端口 *B*；反之，则发送到端口 *D*。

解：

```
#include <P18F458.h>
void main(void)
  {
    unsigned char mybyte;
    TRISC = 0xFF;               //make Port C an input
    TRISB = 0;
    TRISD = 0;                  //both Port B and D as output
    while(1)
      {
        mybyte = PORTC;         //get a byte from PORTC
        if(mybyte < 100)
          PORTB = mybyte;       //send it to PORTB if less than 100
        else
          PORTD = mybyte;       //send it to PORTD if more than 100
      }
  }
```

260

7.2.2 位寻址 I/O 编程

PIC18 的 I/O 端口是可以位寻址的，可以在访问其中的某个位时而不影响其他的位。使用指令 PORTxbits.Rxy 可以访问 Portx 的某个位，其中 x 代表端口 *A*、*B*、*C*、*D*，而 y 代表该端口的位(0～7)。例如，PORTBbits.RB7 表示 PORTB.7。采用同样的方法也可以访问 TRISx 寄存器，如 TRISBbits.TRISB7 表示 TRISB 的 D7 位。表 7-2 列出了 PIC18 的位地址。请学习下面的几个例子，熟悉其中的语法。

表 7-2 PIC18F458/4580 端口的位地址

PORTA	PORTB	PORTC	PORTD	PORTE	端口的位
RA0	RB0	RC0	RD0	RE0	D0
RA1	RB1	RC1	RD1	RE1	D1
RA2	RB2	RC2	RD2	RE2	D2
RA3	RB3	RC3	RD3		D3

（续）

PORTA	PORTB	PORTC	PORTD	PORTE	端口的位
RA4	RB4	RC4	RD4		D4
RA5	RB5	RC5	RD5		D5
	RB6	RC6	RD6		D6
	RB7	RC7	RD7		D7

7.2.3 端口位的结构

图 7-4 展示了 C18 C 编译器给定的端口 *B* 位的结构。在微控制器头文件里也可以找到端口的结构。

```
extern volatile near unsigned char        PORTB;
extern volatile near union {
  struct {
    unsigned RB0:1;
    unsigned RB1:1;
    unsigned RB2:1;
    unsigned RB3:1;
    unsigned RB4:1;
    unsigned RB5:1;
    unsigned RB6:1;
    unsigned RB7:1;
  };
  struct {
    unsigned INT0:1;
    unsigned INT1:1;
    unsigned CANTX:1;
    unsigned CANRX:1;
    unsigned :1;
    unsigned PGM:1;
    unsigned PGC:1;
    unsigned PGD:1;
  };
} PORTBbits;
```

图 7-4　端口 *B* 位结构

例 7-12　编写 C18 程序，连续地翻转 RB4，但不要影响端口 *B* 的其他位。

解：

```
#include <P18F458.h>
#define mybit PORTBbits.RB4          //declare single bit
void main(void)
  {
    TRISBbits.TRISB4=0;              //make RB4 an output
    while(1)
      {
        mybit = 1;                   //turn on RB4
        mybit = 0;                   //turn off RB4
      }
  }
```

例 7-13 编写 C18 程序，监视 PC5。若为高电平，则将数据 55H 发送到端口 B；否则发送 AAH 到端口 D。

解：

```
#include <P18F458.h>
#define mybit PORTCbits.RC5          //notice single-bit declaration
void main(void)
  {
     TRISCbits.TRISC5 = 1;           //RC5 as input
     TRISD = 0;                      //Ports C and D output
     while(1)
      {
        if(mybit == 1)
          PORTD = 0x55;
        else
          PORTD = 0xAA;
      }
  }
```

Address	SFR Name	Hex	Decimal	Binary	Char
0F80	PORTA	00	0	00000000	.
0F81	PORTB	00	0	00000000	.
0F82	PORTC	00	0	00000000	.
0F83					
0F84	PORTE	00	0	00000000	.

Address	SFR Name	Hex	Decimal	Binary	Char
0F80	PORTA	00	0	00000000	.
0F81	PORTB	00	0	00000000	.
0F82	PORTC	20	32	00100000	.
0F83					
0F84	PORTE	00	0	00000000	.

图 7-5 例 7-13 在 MPLAB 仿真器上的运行结果

261 ~ 262

例 7-14 RB1 引脚连有门传感器，RC7 引脚连有蜂鸣器。编写 C18 程序，监视门传感器。当门打开时，蜂鸣器响起，即向蜂鸣器发送频率为几百赫兹的方波。

解：

```
#include <P18F458.h>
void MSDelay(unsigned int);
#define Dsensor PORTBbits.RB1
#define buzzer PORTCbits.RC7
void main(void)
  {
    TRISBbits.TRISB1 = 1;          //PORTB.1 as an input
```

```
    TRISCbits.TRISC7 = 0;          //make PORTC.7 an output
    while(Dsensor == 1)
      {
        buzzer = 0;
        MSDelay(200);
        buzzer = 1;
        MSDelay(200);
      }
    while(1);                  //stay here forever
  }

void MSDelay(unsigned int itime)
  {
    unsigned int i;
    unsigned char j;
    for(i=0;i<itime;i++)
      for(j=0;j<165;j++);
  }
```

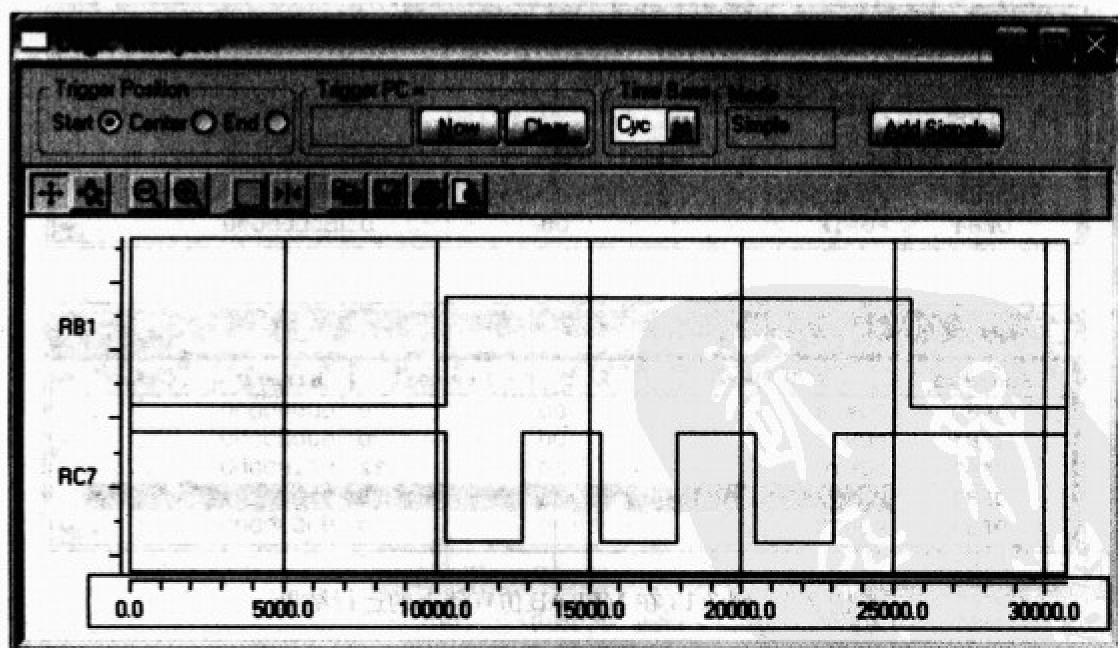


图 7-6 例 7-14 的 MPLAB 逻辑分析器结果

例 7-15 端口 *B* 连接有 LCD 的数据引脚。只要 LCD 的使能端从高电平跳变到低电平，信息就会被锁存到 LCD 中。编写 C18 程序，在 LCD 上显示 The Earth is but One Country。

解:

```
#include <P18F458.h>
#define LCDData PORTB                //LCDData declaration
#define En PORTCbits.RC2             //the Enable pin
```

```
void main(void)
  {
    unsigned char message[] = "The Earth is but One Country";
    unsigned char z;
    TRISB = 0;                          //Port B as output
    TRISCbits.TRISC2 = 0;               //PortC.2 as output
    for(z=0;z<28;z++)                   //send all the 28 characters
      {
       LCDData = message[z];
       En=1;                            //a HIGH-
       En=0;                      //-to-LOW pulse to latch the LCD data
      }
    while(1);                     //stay here forever
  }
```

在仿真器上运行上面的程序，观察如何在 PORTB 将信息的每个字符显示出来。同时，每个字符显示之后，观察 PC.2 的状态。

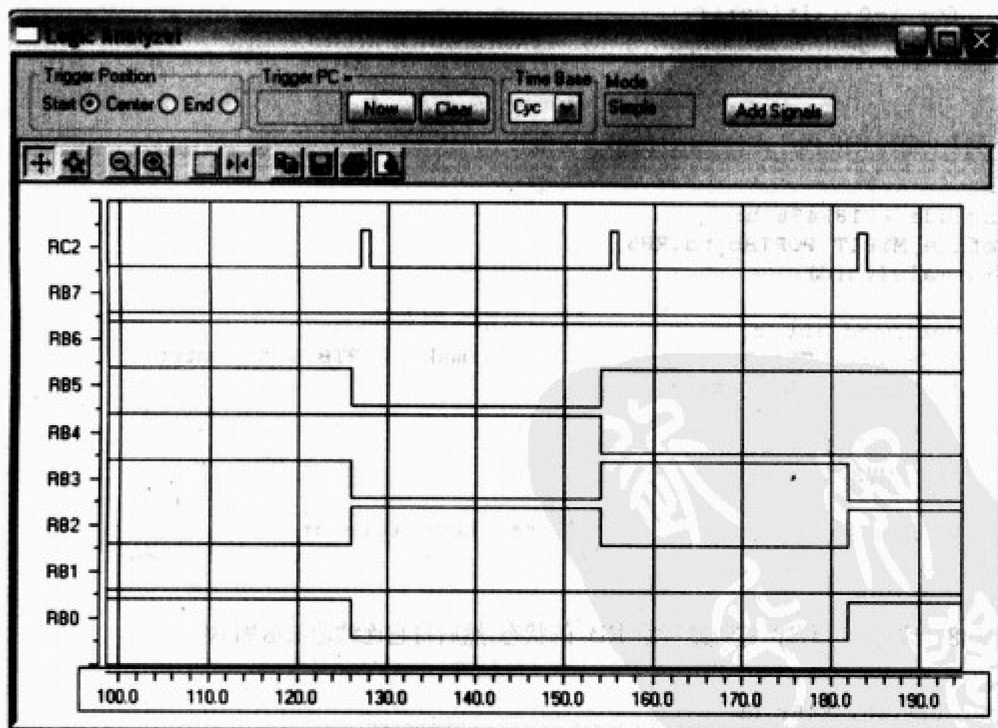


图 7-7 例 7-15 的 MPLAB 逻辑分析器结果

例 7-16 编写 C18 程序，连续地翻转端口 B、C、D，间隔时间为 250 ms。

解：

```
#include <P18F458.h>
void MSDelay(unsigned int);
void main(void)
  {
    TRISB = 0;
```

```
      TRISC = 0;
      TRISD = 0;
      while(1)                        //do it forever
        {
          PORTB = 0x55;
          PORTC = 0x55;
          PORTD = 0x55;
          MSDelay(250);               //250 ms delay
          PORTB = 0xAA;
          PORTC = 0xAA;
          PORTD = 0xAA;
          MSDelay(250);
        }
    }

void MSDelay(unsigned int itime)
    {
      unsigned int i;
      unsigned char j;
      for(i=0;i<itime;i++)
        for(j=0;j<165;j++);
    }
```

例 7-17 编写C18程序,将端口 *B* 的第5位翻转50 000次。

解:

```
#include <P18F458.h>
#define MYBIT PORTBbits.RB5
void main(void)
    {
      unsigned int z;
      TRISBbits.TRISB5 = 0;             //make PORTB.5 an output
      for(z=0;z<50000;z++)
        {
          MYBIT = 1;
          MYBIT = 0;
        }
      while(1);                        //stay here forever
    }
```


例 7-18 编写C18程序,连续地读取RB0的状态,然后将它连续地发送到RC7。

解:

```
#include <P18F458.h>
#define inbit PORTBbits.RB0
#define outbit PORTCbits.RC7
void main(void)
    {
      TRISBbits.TRISB0 = 1;            //make RB0 an input
      TRISCbits.TRISC7 = 0;            //make RC7 an output
      while(1)
        {
         outbit = inbit;               //get a bit from RB0
                                       //and send it to RC7
        }
    }
```

```
1:              #include <P18F458.h>
2:              #define inbit   PORTBbits.RB0
3:              #define outbit  PORTCbits.RC7
4:              void main(void)
5:               {
6:                 TRISBbits.TRISB0 = 1;           //make RB0 an input
0000E2  8093       BSF 0xf93, 0, ACCESS
7:                 TRISCbits.TRISC7 = 0;           //make RC7 an output
0000E4  9E94       BCF 0xf94, 0x7, ACCESS
8:                 while(1)
0000F2  D7F9       BRA 0xe6
9:                   {
10:                    outbit = inbit;             //get bit from RB0
0000E6  5081           MOVF 0xf81, W, ACCESS
0000E8  0B01           ANDLW 0x1
0000EA  E002           BZ 0xf0
0000EC  8E82           BSF 0xf82, 0x7, ACCESS
0000EE  D001           BRA 0xf2
0000F0  9E82           BCF 0xf82, 0x7, ACCESS
                                                   //and send it to RC7
11:
12:                  }
13:             }
0000F4  0012    RETURN 0
```

图 7-8 例 7-18 的反汇编代码

7.2.4 复习题

1. PORTB的地址是________。
2. 编写简短的程序,将端口 C 的所有位翻转。
3. 编写简短的程序,使得端口 B 的第 0 位翻转。
4. 判断对错:PORTB的所有位均可位寻址。
5. 判断对错:TRISB的所有位均可位寻址。

266

7.3 逻辑操作

C语言的一个重要而强大的功能就是具有位操作能力。因为很多C语言的书没有涵盖这部分内容,所以将在这一节讨论它。本节将介绍位的逻辑运算符,并给出一些例子说明它们的用法。

表 7-3 C语言中的位运算符

A	B	AND(逻辑与) A&B	OR(逻辑或) A|B	EX-OR(逻辑异或) A^B	取反 Y=~B
0	0	0	0	0	1
0	1	0	1	1	0
1	0	0	1	1	
1	1	1	1	0	

7.3.1 C语言的位操作符

每个C程序员都很熟悉逻辑操作符:逻辑与(&&)、逻辑或(||)和逻辑非(!),但许多C程序员并不熟悉位操作符,即逻辑与(&)、逻辑或(|)、逻辑异或(^)、取反(~)、右移(≫)和左移(≪)。这些位操作符常用于嵌入式系统和控制的软件工程中,因此,对它们的理解和掌握在基于微控制器的系统和接口设计中显得尤为重要。请参阅表7-3。

下面是几个使用C位操作符的例子。

(1) 0x35 & 0x0F=0x05　　　/* 位逻辑与 */

(2) 0x04 | 0x68=0x6C　　　/* 位逻辑或 */

(3) 0x54 ^ 0x78 =0x2C　　　/* 位逻辑异或 */

(4) ~0x55=0xAA　　　/* 取反 */

7.3.2 C语言的按位移位操作

C语言中有两个移位操作:右移(≫)和左移(≪)。

它们的格式如下:

数据≫要右移的位数

数据≪要左移的位数

下面是C语言中几个使用移位操作符的例子。

(1) 0x9A≫3=0x13　　/* 向右移位3次 */

(2) 0x77≫4=0x07　　/* 向右移位4次 */

(3) 0x6≪4 =0x60　　/* 向左移位4次 */

267

学习例7-19~例7-22,体会怎样在C语言中使用位操作符。

例7-19 在仿真器上运行下面的程序,并查看结果。

解:

```
#include <P18F458.h>
void main (void)
  {
    TRISB = 0;                 //make Ports B, C,
    TRISC = 0;                 //and D output ports
    TRISD = 0;
    PORTB = 0x35 & 0x0F;       //ANDing
    PORTC = 0x04 | 0x68;       //ORing
    PORTD = 0x54 ^ 0x78;       //XORing
    PORTB = ~0x55;             //inverting
    PORTC = 0x9A >> 3;         //shifting right 3 times
    PORTD = 0x77 >> 4;         //shifting right 4 times
    PORTB = 0x6 << 4;          //shifting left 4 times
    while(1);                  //stay here forever
  }
```

例7-20 编写C18程序,连续地翻转端口B和C,间隔时间为250 ms。要求使用取反操作符(~)。

解：

```
#include <P18F458.h>
void MSDelay(unsigned int);
void main(void)
  {
    TRISB = 0;
    TRISC = 0;                //make Ports B and C output
    PORTB = 0x55;
    PORTC = 0xAA;
    while(1)
      {
        PORTB = ~PORTB;
        PORTC = ~PORTC;
        MSDelay(250);
      }
  }
void MSDelay(unsigned int itime)
  {
    unsigned int i;
    unsigned char j;
    for(i=0;i<itime;i++)
      for(j=0;j<165;j++);
  }
```

268

例 7-21 重写 C18 程序，将端口 *B*、*C* 和 *D* 连续地翻转，间隔时间为 250 ms。要求使用异或操作符(^)。

解：

```
#include <P18F458.h>
void MSDelay(unsigned int);
void main(void)
  {
    TRISB = 0;
    TRISC = 0;
    TRISD = 0;                //make Ports B,C, and D output
    PORTB=0x55;
    PORTC=0x55;
    PORTD=0x55;
    while(1)
      {
        PORTB=PORTB^0xFF;
        PORTC=PORTC^0xFF;
        PORTD=PORTD^0xFF;
        MSDelay(250);
      }
  }

void MSDelay(unsigned int itime)
  {
    unsigned int i;
    unsigned char j;
    for(i=0;i<itime;i++)
      for(j=0;j<165;j++);
  }
```

例 7-22 重写 C18 程序，读取 RB0 的状态，将其取反后发送到 RC7。

解：

```
#include <P18F4550.h>
#define inbit PORTBbits.RB0
#define outbit PORTCbits.RC7
void main(void)
  {
     TRISBbits.TRISB0 = 1;          //make PORTB.0 an input
     TRISCbits.TRISC7 = 0;          //make PORTC.7 an output
     while(1)
        {
          outbit = ~inbit;          //get a bit from RB0
        }
  }
```

269

例 7-23 编写 C18 程序，读取 RB0 和 RB1 的状态，并根据下表将对应的 ASCII 字符发送到 PD。

RB1	RB0	
0	0	发送‘0’到 PORTD （注意，‘0’的 ASCII 码为 0x30）
0	1	发送‘1’到 PORTD
1	0	发送‘2’到 PORTD
1	1	发送‘3’到 PORTD

解：

```
#include <P18F458.h>
void main(void)
  {
     unsigned char z;
     TRISB = 0xFF;                  //make Port B an input
     TRISD = 0;                     //make Port D an output
     while(1)                       //repeat forever
        {
          z = PORTB;                //read PORTB
          z = z & 0x3;              //mask the unused bits
          switch(z)                 //make decision
             {
               case(0):
                {
                 PORTD = '0';       //issue ASCII 0
                 break;
                }
               case(1):
                {
                 PORTD = '1';       //issue ASCII 1
                 break;
                }
               case(2):
                {
                 PORTD = '2';       //issue ASCII 2
                 break;
                }
               case(3):
```

```
            {
              PORTD = '3';          //issue ASCII 3
              break;
            }
          }
        }
    }
```

Special Function Registers

Address	SFR Name	Hex	Decimal	Binary	Char
0F80	PORTA	00	0	00000000	.
0F81	PORTB	02	2	00000010	.
0F82	PORTC	00	0	00000000	.
0F83	PORTD	32	50	00110010	2
0F84	PORTE	00	0	00000000	.

270

7.3.3 复习题

1. 下面C程序执行后,请指出每种情况下 PORTB的内容。
   (a) PORTB= 0x37 & 0xCA;
   (b) PORTB= 0x 37 | 0xCA;
   (c) PORTB= 0x 37 ^ 0xCA;
2. 为屏蔽某些位,必须使用________与之做逻辑与操作。
3. 为将某些位置为高电平,必须使用________与之做逻辑或操作。
4. 同自身做逻辑异或操作的结果是________。
5. 下面一段代码执行后,请指出 PORTC 的内容:

```
PORTC = 0;
PORTC = PORTC | 0x99;
PORTC = ~PORTC;
```

7.4 C语言的数据转换程序

回顾第5章和第6章讨论过的BCD数。正如那里提到的,许多新型微控制器均有实时时钟(RTC),即使处于断电的状态也保存着时间和日期。大多数情况下,RTC提供的时间和日期都是压缩BCD码。为了显示它们,就必须将其转换为ASCII码。这一节将介绍逻辑和循环移位指令在BCD码和ASCII码转换中的应用。

7.4.1 ASCII数

在ASCII键盘上,当键0被激活时,就向电脑提供011 0000(30H)。类似地,如果是键1,就提供011 0001(31H),等等。请看表7-4。

表7-4 数字0~9的ASCII码

按键	ASCII(十六进制)	二进制	BCD码(非压缩)
0	30	011 0000	0000 0000
1	31	011 0001	0000 0001
2	32	011 0010	0000 0010
3	33	011 0011	0000 0011
4	34	011 0100	0000 0100
5	35	011 0101	0000 0101
6	36	011 0110	0000 0110
7	37	011 0111	0000 0111
8	38	011 1000	0000 1000
9	39	011 1001	0000 1001

7.4.2 压缩BCD码到ASCII码的转换

不管是供电状态还是断电状态,RTC连续地提供每天的时间(时、分、秒)和日期(年、月、日)。然而,这些数据都是以压缩BCD码的形式提供的。要将其转换为ASCII码,必须首先转换为非压缩BCD码,然后将非压缩BCD码同011 0000(30H)相加。下面将阐述如何将压缩BCD码转换为ASCII码。同时请看例7-24。

压缩BCD码	非压缩BCD码	ASCII码
0x29	0x02, 0x09	0x32, 0x39
00101001	00000010, 00001001	00110010, 00111001

例7-24 编写C18程序,将压缩BCD码0x29转换为ASCII码,并将其发送到PORTB和PORTC用于显示。

解:

```
#include <P18F458.h>
void main(void)
  {
    unsigned char x, y, z;
    unsigned char mybyte = 0x29;
    TRISB = 0;
    TRISC = 0;                    //make Ports B and C output
    x = mybyte & 0x0F;            //mask upper 4 bits
    PORTB = x | 0x30;             //make it ASCII
    y = mybyte & 0xF0;            //mask lower 4 bits
    y = y >> 4;                   //shift it to lower 4 bits
    PORTC = y | 0x30;             //make it ASCII
  }
```

7.4.3 ASCII码到压缩BCD码的转换

要将ASCII码转换为压缩BCD码,必须先将其转换为非压缩BCD码(为了清除3),然后再组合成压缩BCD码。例如,按键4和7分别给出34H和37H,目标是生成压缩BCD码47H(0100 0111B)。

键值	ASCII码	非压缩 BCD 码	压缩 BCD 码
4	34	00000100	
7	37	00000111	01000111 或 47H

转换后处理压缩BCD码,其结果将是压缩BCD格式。第16章将讨论RTC芯片,并使用例7-24和例7-25中所示的BCP和ASCII转换程序。

272

Special Function Registers

Address	SFR Name	Hex	Decimal	Binary	Char
0F80	PORTA	00	0	00000000	.
0F81	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
0F82	PORTC	32	50	00110010	2
0F83	PORTD	00	0	00000000	.
0F84	PORTE	00	0	00000000	.

图 7-9 MPLAB 仿真器运行例 7-24 程序的结果

例 7-25 编写C18程序,将ASCII数"4"和"7"转换成压缩BCD码,并在PORTB显示。

解:

```
#include <P18F458.h>
void main(void)
  {
    unsigned char bcdbyte;
    unsigned char w = '4';
    unsigned char z = '7';
    TRISB = 0;                  //make Port B an output
    w = w & 0x0F;               //mask 3
    w = w << 4;                 //shift left to make upper BCD digit
    z = z & 0x0F;               //mask 3
    bcdbyte = w | z;            //combine to make packed BCD
    PORTB = bcdbyte;
  }
```

File Registers

Address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
04F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0500	00	47	40	07	07	00	00	00	00	00	00	00	00	00	00	00	.G@..... ....
0510	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0520	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0530	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0540	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

Hex Symbolic

Special Function Registers

Address	SFR Name	Hex	Decimal	Binary	Char
0F80	PORTA	00	0	00000000	.
0F81	PORTB	47	71	01000111	G
0F82	PORTC	00	0	00000000	.
0F83	PORTD	00	0	00000000	.
0F84	PORTE	00	0	00000000	.
0F89	LATA	00	0	00000000	.
0F8A	LATB	47	71	01000111	G

图 7-10 MPLAB 仿真器运行例 7-25 程序的结果

273

7.4.4 ROM的校验和

为了保证ROM中数据的完整性,每个系统都必须执行校验和计算。校验和将会检验程序ROM里的数据是否遭到破坏。程序ROM遭到破坏的原因之一就是浪涌电流,在系统开启时或正在运行时都有可能发生。为了确保ROM中数据的完整性,校验和使用了所谓的校验和字节。校验和字节是额外的字节,附加在字节数据的后面。计算字节数据的校验和,可遵循下面的步骤。

(1) 将所有字节数据相加,去掉进位。

(2) 将所得的和做取补运算,得到的就是校验和字节,它是数据序列的最后一个字节。

在执行校验和操作时,将包含校验和字节在内的所有字节相加,查看结果是否为0。如果不是0,数据已被篡改。为理清这些重要的概念,请看例7-26~例7-28。

例7-26 假定有4个十六进制的数据:25H、62H、3FH、52H。

(a) 计算校验和字节数。

(b) 执行校验和操作,验证数据的完整性。

(c) 如果第二个字节62H变成了22H,怎样用校验和方法来探测错误?

解:

```
      25H
    + 62H
(a) + 3FH    丢弃进位1,并做取补运算,得到E8H。
    + 52H
    -----
     118H

      25H
    + 62H
(b) + 3FH    丢弃进位1,得到00,说明数据没有出错。
    + 52H
    + E8H
    -----
     200H

      25H
    + 22H
(c) + 3FH    丢弃进位1,得到C0H,而不是00H,说明数据出错。
    + 52H
    + E8H
    -----
     1C0H
```


例7-27 编写C18程序,计算例7-26所给数据的校验和字节。

解:

```
#include <P18F458.h>
void main(void)
  {
     unsigned char mydata[] = {0x25,0x62,0x3F,0x52};
     unsigned char sum = 0;
     unsigned char x;
     unsigned char chksumbyte;
```

```
    TRISB = 0;
    TRISC = 0;                          //make Ports B and C output
    for(x=0;x<4;x++)
      {
        PORTC = mydata[x];              //issue each byte to PORTC
        sum = sum + mydata[x];          //add them together
        PORTB = sum;                    //issue the sum to PORTB
      }
    chksumbyte = ~sum + 1;              //make 2's complement (invert + 1)
    PORTB = chksumbyte;                 //show the checksum byte
  }
```

在MPLAB中单步执行上面的程序,检查PORTB和PORTC的内容。注意在执行加法运算时,每个字节都放到PORTC中。

例7-28 编写C18程序,执行例7-26的步骤(b)。若数据完好,则发送ASCII字符G到PORTD。否则,发送ASCII字符B到PORTD。

解:

```
#include <P18F458.h>
void main(void)
  {
    unsigned char mydata[] = {0x25,0x62,0x3F,0x52,0xE8};
    unsigned char chksum = 0;
    unsigned char x;
    TRISD = 0;                          //make Port D an output
    for(x=0;x<5;x++)
      chksum = chksum + mydata[x];      //add them together
    if(chksum == 0)
      PORTD = 'G';
    else
      PORTD = 'B';
  }
```

改变数组mydata的一两个值,重新仿真上面的程序,看看结果如何。

275

7.4.5 PIC18二进制(十六进制)到十进制和ASCII的转换

C语言中的printf函数是标准I/O库的一部分,可以利用它做很多事,包括将二进制(十六进制)转换为十进制,反之亦然。但是printf占用了大量的内存空间,并毫无疑问地使你的十六进制文件增大。因此,对基于PIC18微控制器的系统,最好是编写自己的转换函数代替printf函数。

一种很常用的数据转换就是从二进制转换为十进制。在一些器件[如ADC(模数转换器)]中,数据是以二进制的形式提供给微处理器的。有些RTC也提供二进制的时间和日期。为了显示这些二进制数据,首先需要将其转换为十进制,然后转换为ASCII码。因为十六进制表示二进制数比较方便,所以这里所说的二进制,实际上就是十六进制。将二进制数00~FFH转换为十进制即为000~255。实现这种转换的方法是除10取余,这在第5章和第6章介绍过。例如,11111101(FDH)的十进制数是253。下面是实现这种转换的一种算法。

十六进制	商	余数
FD/0A	19	3(低位) LSD
19/0A	2	5(中位)
		2(高位)MSD

例 7-29 给出了这种算法的 C 程序。

例 7-29 编写 C18 程序,将 FDH 转换为十进制,并在 PORTB、PORTC 和 PORTD 上显示。

解:

```
#include <P18F458.h>
void main(void)
  {
     unsigned char x, binbyte, d1, d2, d3;
     TRISB = 0;
     TRISC = 0;
     TRISD = 0;                  //Ports B, C, and D output
     binbyte = 0xFD;             //binary (hex) byte
     x = binbyte / 10;           //divide by 10
     d1 = binbyte % 10;          //find remainder (LSD)
     d2 = x % 10;                //middle digit
     d3 = x / 10;                //most-significant digit (MSD)
     PORTB = d1;
     PORTC = d2;
     PORTD = d3;
  }
```


7.4.6 复习题

1. 将下面的十进制数表示成压缩 BCD 码和非压缩 BCD 码的形式:
   (a)15 (b)99
2. 给出"76"的二进制、十六进制和压缩 BCD 码形式。
3. BCD 码 67H 转换为 ASCII 码是________H 和________H。
4. 下面是将非压缩 BCD 码转换为 ASCII 码吗?
   mydata= 0x09+ 0x30;
5. 为什么用压缩 BCD 码比用 ASCII 码好?
6. 比较压缩 BCD 码与 ASCII 码,哪个占的空间多?
7. 比较压缩 BCD 码与 ASCII 码,哪个更普遍?
8. 计算下列数据的校验和:22H、76H、5FH、8CH、99H。
9. 为检验数据的完整性,将所有字节数相加,包括校验和字节,所得结果必须为________,才能表明数据没被篡改。
10. ADC 的输出为 0010 0110,怎样将其显示在屏幕上?

7.5 C 语言的数据串行化

串行化数据是将一个字节的数据通过微控制器的一个引脚一位一位地发送出去。有两

种方法可用来将字节数据串行地传送。

(1) 使用串行端口。在使用串行端口时，程序员对数据传送顺序的控制受到很大限制。关于串行端口数据传送的内容将在第 10 章讨论。

(2) 传送数据时每次只传一个位，并控制数据的顺序和间隔。在很多新一代器件(如 LCD、ADC 和 EEPROM)上，支持数据串行化的产品越来越受欢迎，因为它们在印制电路板上占用更少的空间。虽然可以用 I^2C、CAN 总线标准，但不是所有的器件都支持这些标准。因此，需要熟悉用 C 语言实现数据串行化。

考察下面的 4 个例子，体会在 C 语言中是怎样实现串行化的。

例 7-30 编写 C18 程序，通过 RC0 串行地发送数据 44H，每次发送一位。首先发送 LSB 位。

解：

```
//Serializing data via RC0 (SHIFTING RIGHT)
#include <P18F458.h>
#define PC0 PORTCbits.RC0
void main(void)
  {
    unsigned char conbyte = 0x44;
    unsigned char regALSB;
    unsigned char x;
    regALSB = conbyte;
    TRISCbits.TRISC0 = 0;           //make RC0 an output
    for(x=0;x<8;x++)
      {
        PC0 = regALSB & 0x01;
        regALSB = regALSB >> 1;
      }
  }
```

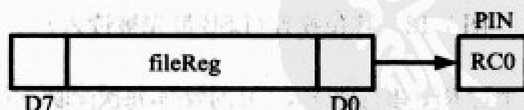


图 7-11 移位操作(LSB 最先被移出)

例 7-31 编写 C18 程序，通过 RC0 串行地发送数据 44H，每次发送一位。首先发送 MSB。

解：

```
//Serializing data via RC0 (SHIFTING LEFT)
#include <P18F458.h>
#define PC0 PORTCbits.RC0
void main(void)
  {
    unsigned char conbyte = 0x88;
    unsigned char regAMSB;
    unsigned char x;
    regAMSB = conbyte;
    TRISCbits.TRISC0 = 0;           //make RC0 an output
    for(x=0;x<8;x++)
```

```
        {
          PC0 = (regAMSB >> 7) & 0x01;
          regAMSB = regAMSB << 1;
        }
    }
```

278

例 7-32 编写 C18 程序,通过 RB0 串行地读入 1 B 的数据,每次读取一位。将读得的该字节数据存放到端口 *D*。首先读入 LSB。

解:

```
//Bringing in data via RB0 (SHIFTING RIGHT)
#include <P18F458.h>
#define PB0 PORTBbits.RB0
void main(void)
  {
    unsigned char x;
    unsigned char REGA=0;
    TRISBbits.TRISB0 = 1;     //RB0 as input
    TRISD = 0;                //Port D as output
    for(x=0;x<8;x++)
      {
       REGA = REGA >> 1;
       REGA |= (PB0 & 0x01) << 7;
      }
    PORTD = REGA;
  }
```

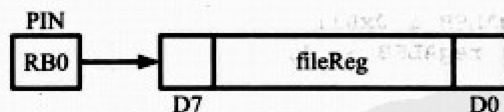


图 7-12 移位操作(LSB 最先被读入)

例 7-33 编写 C18 程序,通过 RB0 串行地读入 1 B 的数据,每次读取一位。首先读入 MSB。

解:

```
//Bringing in data via RB0 (SHIFTING LEFT)
#include <P18F458.h>
#define PB0 PORTBbits.RB0
void main(void)
  {
    unsigned char x;
    unsigned char REGA=0;
    TRISBbits.TRISB0 = 1;     //RB0 as input
    TRISD = 0;                //Port D as output
    for(x=0;x<8;x++)
      {
       REGA = REGA << 1;
       REGA |= PB0 & 0x01;
      }
    PORTD = REGA;
  }
```

279

7.6 C18 程序存储区配置

将预定义的定值数据放入程序(代码)空间是 PIC18 一种常见的做法,正如在第 6 章提到的。第 6 章已经介绍了怎样使用汇编语言指令来访问程序代码空间里的数据。这一章将使用 C18 的 C 编译器来探究同样的问题,同时也会讨论用于 ROM 访问的限定词 far 和 near。

7.6.1 RAM 数据空间与代码数据空间

在 PIC18 中,有以下两个空间可存储数据。

(1) 4096 B 的数据存储区,地址范围是 000~FFFH。在前几章已提到,许多 PIC18 芯片的文件寄存器数据 RAM 都没有 4096 B 那么多,同时还学习到如何直接或间接地读和写 RAM 空间。

(2) 2 MB 的代码(程序)空间,地址范围是 00000~1FFFFFH。这 2 MB 的片上 ROM 空间常用来存放程序代码,因此它受程序计数器(PC)的直接控制。正如在前几章见到的,许多 PIC18 芯片的片上程序 ROM 空间都不到 2 MB。对于存储的数据,也介绍了怎样使用指令 TBLRD 来访问程序 ROM,请参阅第 6 章。使用程序代码框架存放定值数据存在的一个问题是:用于存放数据的空间越多,则留给程序的空间就越少。例如,在只有 4 KB 片上 ROM 的芯片 PIC18F252 中,若用 1 KB 存放查询表,则只有 3 KB 留给程序。对于有些应用,这就会成为一个问题。因此 Microchip 公司向 PIC18 增加了 EEPROM 用于数据存储。PIC18 的 EEPROM 将在第 14 章讨论。下面,将考察 C18 编译器是怎样使用片上 ROM 空间的,以及怎样将数据存入程序 ROM 中。

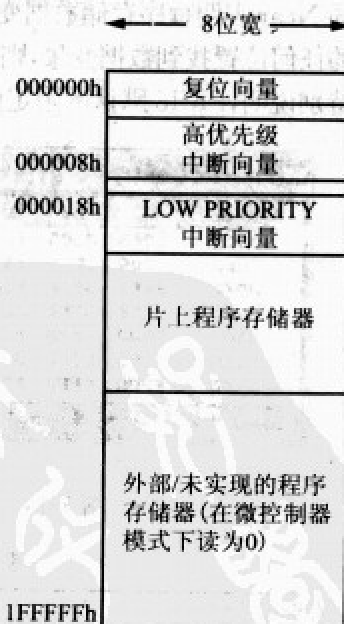


图 7-13 PIC18 程序 ROM 空间

280

表 7-5 PIC18F 系列程序存储区大小

	片上程序 ROM(B)	代码地址范围(十六进制)
PIC18F2220	4 K	00000~00FFF
PIC18F2410	16 K	00000~03FFF
PIC18F458/4580	32 K	00000~07FFF
PIC18F6680	64 K	00000~0FFFF
PIC18F8722	128 K	00000~1FFFF

7.6.2 为数据分配程序空间

到现在为止所讲到的例子中,字节变量都存在数据 RAM 内。在第 6 章介绍过,使用片上 ROM 存储定值数据(如字符串)是一种通常的做法。由于文件寄存器数据 RAM 的空间有限,这种做法就显得特别有用。为使 C18 编译器能为定值数据分配程序(代码)ROM 空间,需要使用关键字 rom,其用法如下面的 C 代码行所示。

```
rom char mynum[] = "Hello";          //use code space for data
rom char weekdays = 7, month= 12;    //use code space for data
```

下面的程序表明在 C18 里怎样使用用于数据存储的程序空间。

程序　7-1

```
//Program 7-1
#include <P18F458.h>
rom const char mynum[]= "0123456789"; //uses program
                                   //ROM space for fixed (constant) data
void main(void)
  {
    unsigned char z;
    TRISB = 0;            //make Port B an output
    for(z=0;z<10;z++)
      PORTB=mynum[z];
  }
```

281

7.6.3　用于程序的 NEAR 与 FAR

前面已讨论过，PIC18 微控制器最多有 2 MB 的片上程序 ROM 空间，但并不是每个型号都有那么多的片上程序 ROM 空间。一些 PIC 只有 4 KB，或者只有 128 KB。为更高效地利用程序空间，C18 编译器允许使用存储限定词 near 和 far 来表征数据和代码的存储区域。限定词 Near 表明程序存储数据变量位于 ROM 的第一个 64 KB 中。若要在 2 MB 的 ROM 空间的任何位置找到数据变量，则要使用限定词 far。请参阅表 7-6 和程序 7-2A。注意，如果没有特别说明，PIC18 默认的限定词是 far。

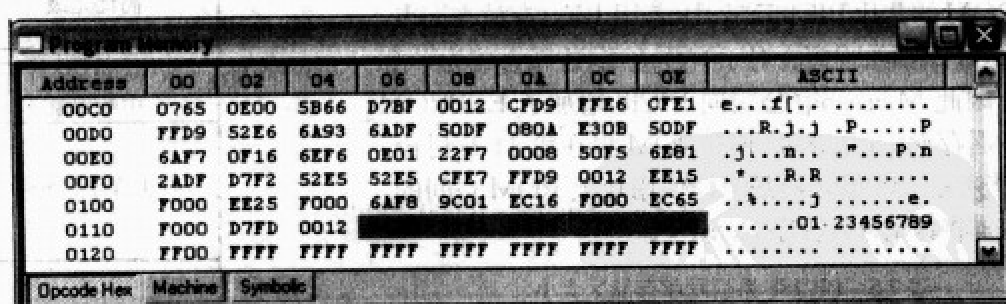


图 7-14　MPLAB 给出的程序 ROM 里的定值数据

表 7-6　用于 ROM 的限定词 near 和 far

存储限定词	ROM 空间
near	程序空间的 0000～FFFFH(64 KB)
far	程序空间的 000000～1FFFFFH(2MB)

程序　7-2A

```
//Program 7-2A
#include <P18F458.h>
   near rom const char mydata[] = "HELLO"; //program ROM data
void main(void)
  {
    unsigned char z;
    TRISB = 0;                          //make Port B an output
    for(z=0;z<5;z++)
```

```
      PORTB = mydata[z];
  }
```

Address	00	02	04	06	08	0A	0C	0E	ASCII
00B0	D7F9	C067	FFF6	C068	FFF7	C069	FFF8	0100	..g...h. ..i.....
00C0	0765	0E00	5B66	D7BF	0012	CFD9	FFE6	CFE1	e...f[.. ....
00D0	FFD9	52E6	6A93	6ADF	50DF	0805	E30B	50DF	...R.j.j .P.....P
00E0	6AF7	0F16	6EF6	0E01	22F7	0008	50F5	6E81	.j...n.. ."...P.n
00F0	2ADF	D7F2	52E5	52E5	CFE7	FFD9	0012	EE15	.*...R.R ....
0100	F000	EE25	F000	6AF8	9C01	EC16	F000	EC65	..%....j ......e.
0110	F000	D7FD	0012						.......HE LLO.....

Opcode Hex　Machine　Symbolic

注意一下用于地址的 4 位数(0000～FFFF)。

图 7-15　MPLAB 给出的使用存储限定词 near 的情况

282

在程序 7-2A 中,如果将 near 改为 far,然后在 PIC18F8722 芯片(带有 128KB 的程序 ROM)上编译,有下面的程序:

程序　7-2B

```
//program 7-2B
#include <P18F8722.h>
    far rom const char mydata[]= "HELLO"; //program ROM data
void main(void)
  {
    unsigned char z;
    TRISB = 0;                                //make Port B an output
    for(z=0;z<5;z++)
      PORTB = mydata[z];
  }
```

Address	00	02	04	06	08	0A	0C	0E	ASCII
000D0	FFD9	52E6	6A93	6ADF	50DF	0805	E31C	0E38	...R.j.j .P....8.
000E0	6E00	0E01	6E01	0E00	6E02	CFDF	F003	6A04	.n...n.. .n.....j
000F0	6A05	5000	2403	6EF3	5001	2004	6EF4	5002	.j.P.$.n .P. .n.P
00100	2005	6EF8	CFF4	FFF7	CFF3	FFF6	0008	50F5	. .n.... ....P
00110	6E81	2ADF	D7E1	52E5	52E5	CFE7	FFD9	0012	.n.*...R .R......
00120	EE1E	F000	EE2E	F000	6AF8	9C07	EC16	F000	.... .j......
00130	EC65	F000	D7FD	0012	[illegible]	[illegible]	[illegible]	FFFF	e....... HELLO...

Opcode Hex　Machine　Symbolic

注意一下用于地址的 5 位数(00000～FFFFF)。

图 7-16　MPLAB 给出的使用存储限定词 far 的情况

7.6.4　Pragma 和数据与程序的固定地址分配

在第 6 章已经介绍过,MPLAB 汇编器允许用 ORG 伪指令将数据或程序放在指定的 ROM 地址中。在 C18 C 编译器中要实现同样的功能,需要使用伪指令 #pragma section,其中 section 是程序或数据的一部分,可以为它们分配特定的存储地址。对于片上 ROM 程序存储器,有两种方法可供选择:(1)code,(2)romdata。伪 #pragma 指令因包含可执行指令而用于程序代码;而伪 #pragma romdata 指令则用于定值数据,如字符串和查询表。下面来探讨使用伪 #pragma 指令为程序和数据分配 ROM 地址。

7.6.5　在指定的 ROM 地址放置代码

为了将程序(包含可执行的指令)放到程序 ROM 的指定地址,可以使用伪指令#pragma 283 code。考察程序 7-3,观察函数 MSDelay 的 C 代码是怎样放置在 ROM 地址 0x300 中的。

程序　7-3

```
//Program 7-3
#include <P18F458.h>
#pragma code main = 0x50        //place the main at ROM addr 0x50
void MSDelay(unsigned int);
void main(void)
  {
    unsigned char mydata[] = "HELLO";
    unsigned char z;
    TRISB = 0;                  //make Port B an output
    for(z=0;z<5;z++)
     {
       PORTB = mydata[z];
       MSDelay(250);
     }
  }

#pragma code MSDelay = 0x300  //place delay at ROM addr 0x300
void MSDelay(unsigned int itime)
  {
    unsigned int i;
    unsigned char j;
    for(i=0;i<itime;i++)
      for(j=0;j<165;j++);
  }
```

284 在 MPLAB 上运行上面的程序,检查程序代码空间,可以发现 main 和 MSDelay 函数分别位于 ROM 地址 0x50 和 0x300。

7.6.6　在指定的 ROM 地址放置代码

为了将数据(包括变量、常量、字符串、查询表)放到程序 ROM 的指定地址,可以使用伪指令#pragma romdata。考察程序 7-4,观察 C 代码是如何将程序 ROM 地址 0x200 分配给字符串 Hello 的。

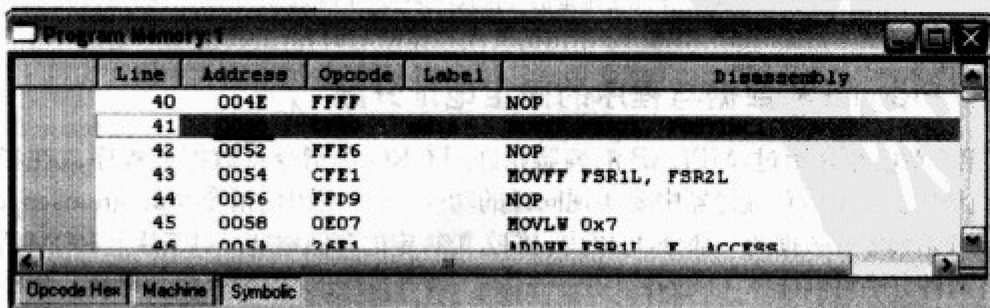


图 7-17　程序 7-3 的屏幕截图

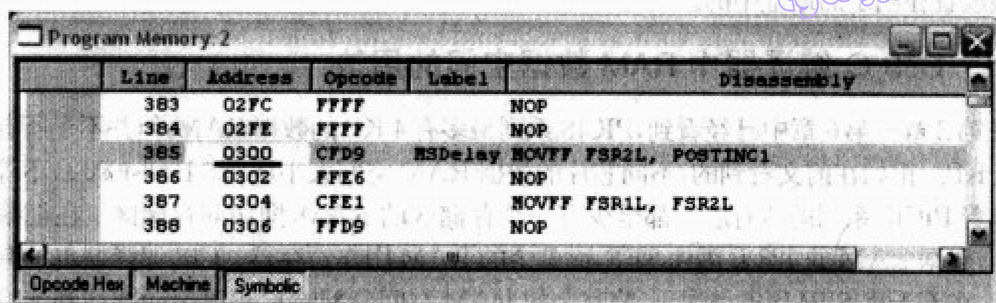


图 7-17 （续）

程序 7-4

```
//Program 7-4
#include <P18F458.h>
#pragma romdata mydata = 0x200  //place mydata at ROM addr 0x200
    near rom const char mydata[]= "HELLO";    //ROM data
void main(void)
  {
    unsigned char z;
    TRISB = 0;                    //make Port B an output
    for(z=0;z<5;z++)
       PORTB = mydata[z];
  }
```

在 MPLAB 上运行上面的程序。检查程序代码空间，可以看到字符串 Hello 位于起始地址为 0x50 的 ROM 地址中。

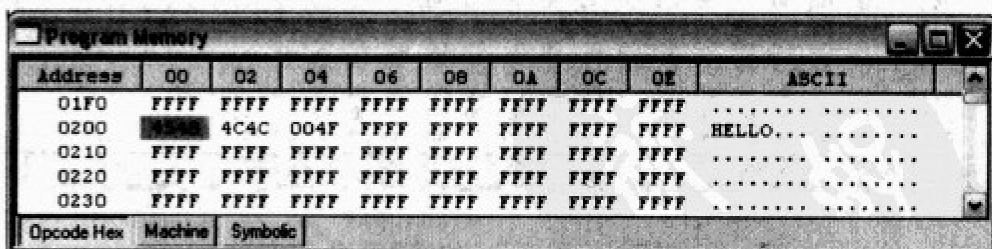


图 7-18 程序 7-4 的屏幕截图

7.6.7 复习题

1. PIC18 系列最多有________的程序 ROM 空间。
2. PIC18F8722 有________的程序 ROM。
3. 判断对错：程序(代码)ROM 空间可用来存储数据，但数据空间并不能用来存储程序代码。
4. 判断对错：用程序 ROM 空间存储数据意味着数据是固定的、静态的。
5. 如果有超过 1000 B 的消息字符串，那么应该使用________(程序 ROM，数据 RAM)来存放它。

7.7 C18 的数据 RAM 分配

这一节将探讨数据 RAM 文件寄存器的用法，以及 C18 编译器如何对它进行地址分配。还会介绍用于数据 RAM 的存储限定词 near 和 far。另外，还将考察 C18 编译器是怎样将数

据和栈放置在固定的地址中的。

7.7.1　C18 C 编译器中 RAM 数据空间的用法

在第 2 章～第 6 章中已经看到，PIC18 系列最多有 4 KB 的数据 RAM，但并不是所有型号都有 4 KB。正如在前文看到的，不同芯片的数据 RAM 空间大小从 256 B 到 4096 B 不等。这就意味着 PIC18 系列的所有芯片都至少有一个存储区的 RAM，即访问存储区。在汇编程序设计中，数据 RAM 的 128 B 用作 SFR，而其余的 RAM 用作暂存器。C18 编译器也具有同样的功能，在不干扰 SFR 区的情况下，将其余的 RAM 空间分配给 C 程序声明的栈和变量。请参阅程序 7-5。

程序　7-5

```
//Program 7-5
#include <P18F458.h>
void main(void)
 {
    unsigned char x=5,y=9;    //uses data RAM to store data
    unsigned char z;
    TRISB = 0;                //make Port B an output
    z = x + y;
    PORTB = z;
  }
```

在 MPLAB 中运行上面的程序，观察数据 RAM 空间中 x、y 和 z 的位置，如图 7-19 所示。

File Registers

Address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
04D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
04E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
04F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
	00	05	09	0E	0E	00	00	00	00	00	00	00	00	00	00	00	....
0510	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0520	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0530	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

Hex　Symbolic

图 7-19　程序 7-5 的屏幕截图

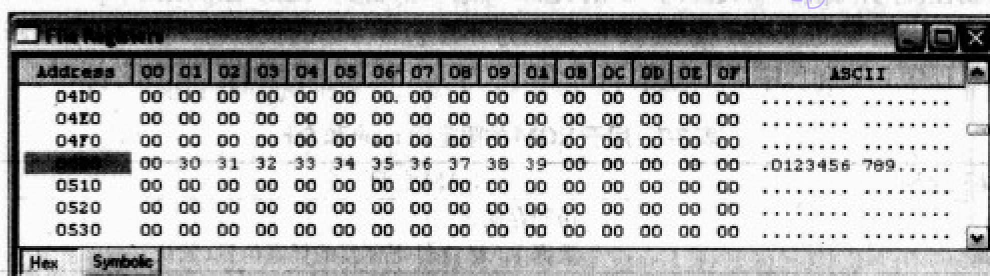
一个数组需要连续的内存空间，用于存放数组元素。也就是说，数组大小要受到 PIC18 芯片数据 RAM 大小的限制。请参阅下面的程序 7-6。

程序　7-6

```
//Program 7-6
#include <P18F458.h>
void main(void)
  {
    unsigned char mynum[]= "0123456789"; //uses RAM space
                                         //to store data
    unsigned char z;
    TRISB = 0;                           //make Port B an output
    for(z=0;z<10;z++)
       PORTB = mynum[z];
  }
```

在 MPLAB 中运行上面的程序，观察数据 RAM 空间，可以找出 30H、31H、32H、……、

41H、42H、43H、44H 等(ASCII 码 0、1、2 等的十六进制,如图 7-20 所示)的 RAM 地址。



File Registers

Address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
04D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
04E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
04F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
[illegible]	00	30	31	32	33	34	35	36	37	38	39	00	00	00	00	00	.0123456 789.....
0510	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0520	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0530	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

Hex Symbolic

图 7-20　程序 7-6 的屏幕截图

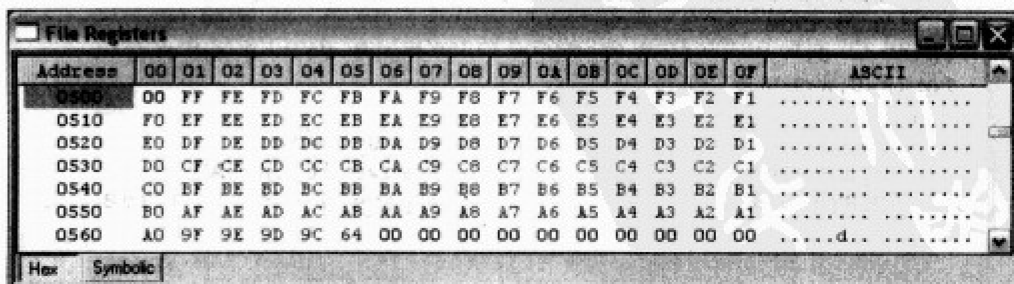
数据具有大量数组元素(如数组长度为 100)的情况处理,请看下面的程序。

程序　7-7

```
//Program 7-7
#include <P18F458.h>
void main(void)
  {
    unsigned char mydata[100];     //100-byte space in RAM
    unsigned char x, z = 0xFF;
    TRISB = 0;                     //make Port B an output
    for(x=0;x<100;x++)
      {
        mydata[x] = z;             //save it in RAM
        PORTB = z;                 //give a copy to PORTB too
        z--;                       //count down
      }
  }
```

在 MPLAB 中运行上面的程序。可以在数据 RAM 文件存储器中找到数值 FFH、FEH、FDH 等的地址,如图 7-21 所示。

287



File Registers

Address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
0500	00	FF	FE	FD	FC	FB	FA	F9	F8	F7	F6	F5	F4	F3	F2	F1	....
0510	F0	EF	EE	ED	EC	EB	EA	E9	E8	E7	E6	E5	E4	E3	E2	E1	....
0520	E0	DF	DE	DD	DC	DB	DA	D9	D8	D7	D6	D5	D4	D3	D2	D1	....
0530	D0	CF	CE	CD	CC	CB	CA	C9	C8	C7	C6	C5	C4	C3	C2	C1	....
0540	C0	BF	BE	BD	BC	BB	BA	B9	B8	B7	B6	B5	B4	B3	B2	B1	....
0550	B0	AF	AE	AD	AC	AB	AA	A9	A8	A7	A6	A5	A4	A3	A2	A1	....
0560	A0	9F	9E	9D	9C	64	00	00	00	00	00	00	00	00	00	00	.....d.. ....

Hex Symbolic

图 7-21　程序 7-7 的屏幕截图

修改数组的长度和目标 PIC18 芯片的型号(如 PIC18F252),使用 C 编译器查看 RAM 空间分配情况。

7.7.2　用于数据的 near 与 far

C18 编译器有两个用于数据 RAM 分配的存储限定词,即 near 和 far。限定词 near 和 far 通常用来表征声明的变量所使用的数据 RAM 地址段。关键词 near 将用于数据声明的 RAM

地址限制在访问存储区，而关键词 far 则表征 C 编译器任意分配的全部数据 RAM 空间。请参阅表 7-7。对于数据 RAM 较小的 PIC18，程序不能包含太多的长数组。使用 C18 编译器，编译和仿真程序 7-8a 和程序 7-8b，观察 near 和 far 对 RAM 地址的影响。请参阅表 7-7。

表 7-7 用于 ROM 的限定词 near 和 far

存储限定词	RAM 空间
near	访问存储区
far	数据 RAM 文件寄存器的任意位置(默认)

程序 7-8a

```
//Program 7-8a
#include <P18F458.h>
near unsigned char mydata[100];         //100-byte space in RAM
void main(void)
  {
     unsigned char x, z = 0;
     TRISB = 0;                         //make Port B an output
     for(x=0;x<100;x++)
       {
         z--;                           //count down
         mydata[x] = z;                 //save it in RAM
         PORTB = z;                     //give a copy to PORTB too
       }
  }
```


程序 7-8b

```
//Program 7-8b
#include <P18F458.h>
void main(void)
  {
     far unsigned char mydata[100];     //100-byte space in RAM
     unsigned char x, z = 0;
     TRISB = 0;                         //make Port B an output
     for(x=0;x<100;x++)
       {
         z--;                           //count down
         mydata[x] = z;                 //save it in RAM
         PORTB = z;                     //give a copy to PORTB too
       }
  }
```

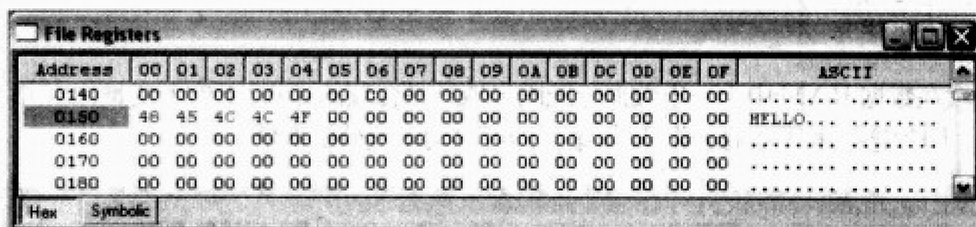
7.7.3 在指定内存地址存放数据

在第 6 章已知道，MPLAB 汇编器允许将数据放到指定的 RAM 地址，只要结合使用 MOVLW 和 MOVWF 即可。在 C18 C 编译器中，要将数据放到指定的 RAM 地址，可以使用伪指令 #pragma。在上一节使用了伪指令 #pragma 来设定 ROM 存储地址，同样，伪指令 #pragma 也可用来设定数据 RAM 地址。当伪用指令 #pragma 设定数据 RAM 地址时，有两种选择：idata 和 udata。其中，idata 代表初始化数据，udata 代表未初始化数据。在 C18 中，idata(初始化的数据)和 udata(未初始化的数据)常用来在数据 RAM 中分配具体的地址。例如，下面的代码使用 idata 将字符串 HELLO 存放到起始地址为 0x150 的 RAM 中。

程序 7-9

```
//Program 7-9 (using idata)
#include <P18F458.h>
#pragma idata mydata = 0x150
unsigned char mydata[]= "HELLO";                //RAM data
void main(void)
  {
    unsigned char z;
    TRISB = 0;                                  //make Port B an output
    for(z=0;z<5;z++)
      PORTB = mydata[z];
  }
```

可以通过仿真上面的程序，来验证上面的一些概念，并检查 RAM 地址 0x150。



File Registers

Address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
0140	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0150	48	45	4C	4C	4F	00	00	00	00	00	00	00	00	00	00	00	HELLO... ....
0160	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0170	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0180	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

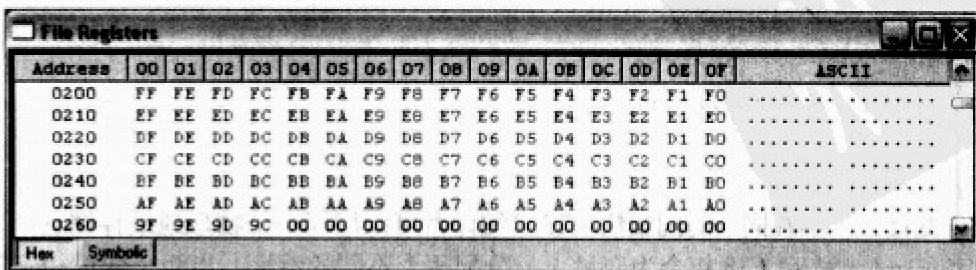
Hex　Symbolic

图 7-22　程序 7-9 的屏幕截图

下面的程序是对早期程序的一个重复，说明了如何使用 udata 分配固定的地址 0x200。

程序 7-10

```
//Program 7-10 (using udata)
#include <P18F458.h>
#pragma udata mycount = 0x200          //assign RAM address 0x200
                                       //(bank 2)
far unsigned char mycount[100];        //100-byte space in RAM
void main(void)
  {
    unsigned char x,z=0;
    TRISB = 0;.                        //make Port B an output
      for(x=0;x<100;x++)
        {
          z--;                         //count down
          mycount[x]=z;                //save it in RAM
          PORTB=z;                     //give a copy to PORTB too
        }
  }
```



File Registers

Address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
0200	FF	FE	FD	FC	FB	FA	F9	F8	F7	F6	F5	F4	F3	F2	F1	F0	....
0210	EF	EE	ED	EC	EB	EA	E9	E8	E7	E6	E5	E4	E3	E2	E1	E0	....
0220	DF	DE	DD	DC	DB	DA	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	....
0230	CF	CE	CD	CC	CB	CA	C9	C8	C7	C6	C5	C4	C3	C2	C1	C0	....
0240	BF	BE	BD	BC	BB	BA	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0	....
0250	AF	AE	AD	AC	AB	AA	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0	....
0260	9F	9E	9D	9C	00	00	00	00	00	00	00	00	00	00	00	00	....

Hex　Symbolic

图 7-23　程序 7-10 的屏幕截图

下面的程序将展示如何同时使用 udata 和 idata 分配固定地址的情况。

程序 7-11

```
//Program 7-11 (assigning udata and idata to a fixed address)
#pragma idata x = 0x100   //assign fixed RAM address 0x100 to var x
unsigned char x=5;        //both data are initialized data
#pragma idata y = 0x101   //assign fixed RAM address 0x101 to var y
unsigned char y=9;        //both data are initialized data
#pragma udata z = 0x102   //assign fixed RAM address 0x102 to var z
unsigned char z;          //it is uninitialized data

#include <P18F458.h>
void main(void)
  {
    TRISB = 0;            //make Port B an output
    z = x + y;
    PORTB = z;
  }
```

虽然在数据 RAM 中为一串数据分配固定地址是合理的，但对于单个变量而言并不提倡这么做。因为这是编译器的工作范畴，编译器可以动态地分配地址。

290

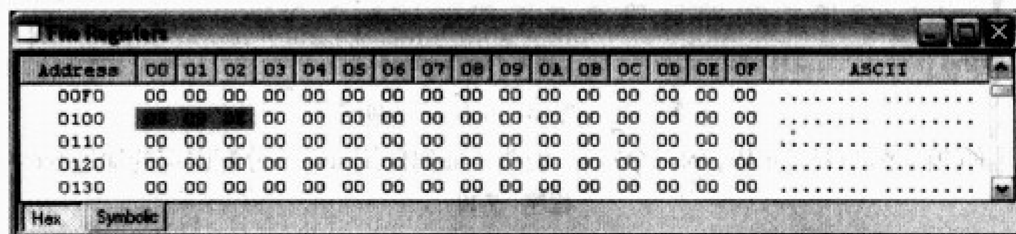


图 7-24 程序 7-11 的屏幕截图

7.7.4 覆盖存储类

为了更高效地使用 PIC18 的数据空间，C18 编译器引入了覆盖存储类(overlay storage class)。Overlay 允许两个变量共享同一个物理地址，只要它们不在同一时间激活。因此 overlay可以节约内存空间。比较下面的两个函数。

```
unsigned char proga(void)
  {
    overlay unsigned char x = 0;
    x = x + 1;
    return x;
  }
```

和

```
unsigned char progb(void)
  {
    overlay unsigned char y = 0;
    y = y + 2;
    return y;
  }
```

由于变量 x、y 不在同一时间激活，C18 C编译器可以用同一个物理地址存储它们。如果将关键字 *overlay* 移除，C18 将会为变量 x、y 分配两个不同的地址。当两个变量相互依存且都处于活动状态时，C18 将会为它们分配两个不同的物理空间。请看下面的情形。

```
unsigned char progc(void)
  {
    overlay unsigned char x = 0;
    x = progd()
    return x;
  }
```

和

```
unsigned char progd(void)
  {
    overlay unsigned char y = 0;
    y = y + 2;
    return y;
  }
```

291

在前一个程序中，尽管使用了关键字 *overlay*，可是 C18 编译器仍然为 x 和 y 分配不同的 RAM 地址。这是因为在函数 progc 中调用了函数 progd，变量之间相互依存并在同一时间处于活动状态。注意，C18 C 编译器支持所有 ANSI C 标准存储类，如 auto、extern、static 等。Overlay 是一个新的存储类，对局部变量有效。

为了更好地理解覆盖存储类的概念，请参阅例 7-34。它给出了 3 个不同版本的程序，将字符串 HELLO 发送到端口 B。使用 C18 编译器，对每个程序进行仿真，比较各种数据存储方法。同时，比较生成的十六进制文件的大小，观察各种数据存储方法对十六进制文件大小的影响。

292

例 7-34 对比下面的 3 个程序，讨论每个程序的优缺点。

解：

(a)

```
#include <P18F458.h>
void main(void)
  {
    TRISB = 0;                    //make Port B an output
    PORTB = 'H';
    PORTB = 'E';
    PORTB = 'L';
    PORTB = 'L';
    PORTB = 'O';
  }
```

(b)

```
#include <P18F458.h>
void main(void)
  {
    unsigned char mydata[] = "HELLO";
    unsigned char z;
    TRISB = 0;                    //make Port B an output
    for(z=0;z<5;z++)
      PORTB = mydata[z];
  }
```

(c)

```
#include <P18F458.h>
void main(void)
  {
    rom unsigned char mydata[] = "HELLO";     //notice keyword rom
    unsigned char z;
    TRISB = 0;                           //make Port B an output
    for(z=0;z<5;z++)
      PORTB = mydata[z];
  }
```

解:

所有程序以不同的方法将字符串 HELLO 发送到端口 B,每次发送一个字节。第一种方法比较简洁,但每个字符都嵌入在程序中。如果要改变某个字符,整个程序就随之改变。这种方法将程序和数据混合在一起了。第二种方法用 RAM 数据空间来存储数组元素,因此数组的大小受文件寄存器大小的限制。第三种方法用程序存储空间的一个单独的区域来存放数据。这就允许你随便设置数组的大小,只要芯片有足够的片上 ROM。然而,在程序 ROM 中,用于数据存储的空间越多,留给程序代码的空间就越少。如果想改变字符串的值,或是想增加其长度,程序(b)和(c)都很容易做到,可是程序(a)就比较困难。

293

7.7.5 复习题

1. PIC18 最多有________数据 ROM。
2. PIC1F8722 有________数据 RAM 空间。
3. 判断对错:数据空间可用来存储程序代码。
4. 在 C18 中,如果要声明下面的变量,你会使用哪个存储空间?
   (a) 一个星期的天数
   (b) 一年的月数
   (c) 延时计数器
5. 判断对错:存储限定词 near 用来将变量存放到访问 RAM 中。

> 访问下面的网站,下载 PIC18C 编译器:http://www.microchip.com
> 下面的网站提供 MPLAB 和 C18 的使用指南:http://www.MicroDigitalEd.com
> 在 PIC18F 硬件设备上运行 C18 程序,需要注意下面两点:
> (1) 在配置位里关闭看门狗定时器。
> (2)在程序的结尾处加上 while(1);防止程序重新执行一遍。这相当于汇编语言里的指令 HERE BRA HERE。

小结

本章介绍了 C18 编程,特别讲解了 I/O 端口的 C 编程和时延程序的 C 实现。同时,介绍了逻辑操作符 AND、OR、XOR 和取补运算。另外,还讨论了这些操作符的一些应用。本章描述了 BCD 码和 ASCII 码的格式,以及它们在 C 中的相互转换;也比较和对比了在 C 语言中的程序代码空间和 RAM 数据空间的用法,还讨论了一项广为使用的技术——数据串行化。

294

习题

1. 对于下面的变量,你会使用哪种数据类型:
   (a) 温度 (b) 一个星期的天数
   (c) 一年的天数 (d) 一年的月数
   (e) 上车人数的计数值 (f) 上课人数的计数值
   (g) 64 KB RAM空间的地址 (h) 一个人的年龄
   (i) 欢迎客人来到大厦的欢迎辞
2. 对于下面的C语句,请指出发送到端口的十六进制数。
   (a) PORTB= 14; (b) PORTB= 0x18; (c) PORTB= 'A';
   (d) PORTB= 7; (e) PORTB= 32; (f) PORTB= 0x45;
   (g) PORTB= 255; (h) PORTB= 0x0F。
3. 请指出影响PIC18微控制器时延代码大小的两个因素。
4. 在第3题的两个因素中,哪个因素可以由系统设计者设置?
5. 程序员可以设置执行一条指令的时钟周期数吗?为什么?
6. 为什么不同的C编译器会产生不同大小的十六进制文件?
7. 请指出PORTBbits.RB4与TRISBbits.TRISB4的区别。
8. 编写C18程序,每200 ms将PORTB的所有位翻转一次。
9. 编写C18程序,每200 ms将RB1位和RB7位翻转一次。
10. 编写一个时延100 ms的函数。
11. 编写C18程序,每200 ms仅将RB0位翻转一次。
12. 编写C18程序,从0~99连续地计数PORTB。
13. 对于下面的指令,指出每个端口的数据。
    注意:每个操作都是独立的。
    (a) PORTB=0xF0&0x45; (b) PORTB=0xF0&0x56;
    (c) PORTB=0xF0^0x76; (d) PORTC=0xF0&0x90;
    (e) PORTC=0xF0^0x90; (f) PORTC=0xF0|0x90;
    (g) PORTC=0xF0&0xFF; (h) PORTC=0xF0|0x99;
    (i) PORTC=0xF0^0xEE; (j) PORTC=0xF0^0xAA;
14. 在下面操作执行后,指出端口的内容。
    (a) PORTB=0x65&0x76; (b) PORTB=0x70|0x6B;
    (c) PORTC=0x95^0xAA; (d) PORTC=0x5D&0x78;
    (e) PORTC=0xC5|0x12; (f) PORTD=0x6A^0x6E;
    (g) PORTB=0x37|0x26;
15. 在下面操作执行后,指出端口的内容。
    (a) PORTB=0x65>>2; (b) PORTC=0x39<<2;
    (c) PORTB=0xD4>>3; (d) PORTB=0xA7<<2;
16. 编写C18程序,将0x95交换成0x59。
17. 编写C18程序,找出一个8位数据中零的个数。
18. 一台步进电机采用下面的二进制序列驱动马达,在C18中你会怎样产生这组序列?
    1100,0110,0011,1001
19. 编写程序,将下面的压缩BCD数转换成ASCII码。假定压缩BCD码位于数据RAM中。

76H,87H,98H,43H

20. 编写程序,将下面的ASCII数转换成压缩BCD数。假定ASCII数位于数据RAM中。
"8767"
21. 编写程序,从PORTB读入8位二进制数,将其转换为ASCII码。如果输入是00~0x99的压缩BCD码,就保存结果。假定PORTB的输入是1000 1001B(二进制)。
22. 对于下面的变量,你会使用哪种存储类型(数据RAM或者程序ROM空间)。
(a) 温度 (b) 一周的天数 (c) 一年的天数 (d) 一年的月数
23. 判断对错:用程序ROM存放数据时,数组的大小不能超过256。
24. 为什么用ROM程序空间来存放视频游戏的文字和形状呢?
25. 用程序ROM空间来存放数据的好处是什么?
26. 用程序ROM空间来存放数据的缺点是什么?
27. 编写C18程序,将一个英文名字的名和姓发送到PORTC。使用程序ROM空间来存放数据。
28. far和near存储之间有什么不同?
29. 伪指令#pragma code与#pragma romdata之间有什么不同?
30. 编写程序,将第27题中的名放在ROM地址0x220,姓放在ROM地址0x200。
31. 指出下列各芯片的程序ROM空间的大小。
(a) PIC18F452/4520 (b) PIC18F458/4580 (c) PIC18F8722

296

32. 在第31题中,讨论ROM空间是如何影响给数据存储分配空间的。
33. 对于下面的变量,你会使用哪种存储类型(数据RAM或者程序ROM空间)?
(a) 上车人数的计数器 (b) 上课人数的计数器
(c) 64 KB RAM空间的地址 (d) 一个人的年龄
(e) 欢迎客人来到大厦的欢迎辞
34. 指出下面芯片的数据RAM空间的大小:
(a) PIC18F452/4520 (b) PIC18F458/4580 (c) PIC18F8722
35. 为什么数组的大小应尽量限制在256B以内?
36. 为什么不用数据RAM空间来存放视频游戏的文字和形状?
37. 使用数据RAM空间来存放固定数据的缺点是什么?
38. 使用数据RAM空间来存放变量的优点是什么?
39. 指令 #pragma idata与#pragma udata之间有什么不同?
40. 编写程序,将第27题的名字放在RAM地址0x300中。
41. 请解释什么时候可以对变量使用overlay。
42. 判断对错:overlay用于那些不在同一时间处于活动状态的变量。

复习题答案

7.1节

1. 无符号字符型0~255;有符号字符型-128~+127。
2. 无符号整型0~65 535;有符号整型-32 768~+32 767。
3. 无符号字符型。
4. 正确。
5. (a) PIC18系统的晶振频率 (b) PIC18机器周期时间 (c) C编译器

7.2 节

1. F81H。
2.
```
void main()
  {
  TRISC = 0;
  PORTC = 0x55;
  PORTC = 0xAA;
  }
```
3.
```
#define PB0bit PORTBbits.RB0
void main()
  {
  TRISBbits.TRISB0 = 0;
  PB0bit = 0;
  PB0bit = 1;
  }
```
4. 正确。 5. 正确。

7.3 节

1. (a) 02 (b) FFH (c) FDH 2. 零 3. 1 4. 全 0 5. 66H

7.4 节

1. (a) 15H=0001 0101 压缩 BCD,0000 0001 0000 0101 非压缩 BCD
   (b) 99H=1001 1001 压缩 BCD,0000 1001 0000 1001 非压缩 BCD
2. 3736H=0011 0111 0011 0110B;压缩 BCD76H=0111 0110B
3. 36,37 4. 是的,因为 mydata=0x39。
5. 节省空间 6. ASCII 7. BCD 8. E4H 9. 0
10. 首先,将二进制转换为十进制,然后转换为 ASCII,将结果传送到屏幕上,可以看到 038。

7.6 节

1. 2 M 2. 128 3. 正确。 4. 正确。 5. 程序 ROM

7.7 节

1. 4 K 2. 4096B 3. 错误。 4. (a) ROM 空间 (b) ROM 空间 (c) RAM 空间
5. 正确。

第 8 章 PIC18F 硬件连接与 ROM 程序载入

学习目标：

- ☐ PIC18F 微控制器复位引脚的功能
- ☐ PIC18F 芯片的硬件连接
- ☐ 晶体振荡器在时钟电路中的应用
- ☐ 基于 PIC18F 的系统设计
- ☐ 掉电复位电压在系统复位中的作用
- ☐ CONFIG 寄存器在 PIC18 系统中的作用
- ☐ PIC Trainer 的设计
- ☐ PIC18 测试程序的编写
- ☐ 使用 PICkit 2 的 PIC18F 程序下载
- ☐ 32 位和 16 位地址的 Intel 十六进制文件特性

299

本章将介绍 PIC18F 系统的物理连接和测试过程。8.1 节介绍 PIC18F458 的引脚功能。8.2 节将介绍 PIC18 的配置寄存器以及如何设置它们。8.3 节将会阐述由 MPLAB 生成的 Intel 十六进制文件的特征。8.4 节将讨论微控制器程序装载的各种方法。本章还介绍采用 PIC18F452/458 (PIC18F4520/4580)芯片的 PIC18 Trainer 的硬件连接。

8.1 PIC18F452/458 的引脚连接

PIC18F458 系列使用不同的封装，如 DIP(双列直插式封装)、QFP(四方扁平封装)和 LLC(无引线芯片载体封装)。它们都有许多的专用于各种各样功能(如 I/O、ADC、定时器和中断)的引脚。为了适用于需求较少的应用，Microchip 公司提供了简化 I/O 端口的 18 引脚芯片。然而，鉴于大多数开发者都使用 40 引脚的芯片，在这里将着重介绍它。图 8-1 是 PIC18F458 的引脚图。

300

观察图 8-1，在芯片的 40 个引脚中，有 33 个引脚是分配给 5 个端口 A、B、C、D、E 的，并具有第二功能。其余的引脚被指定为 V_{dd}、GND(V_{ss})、OSC1、OSC2 和 MCLR(主清零复位)。接下来将逐一地描述各引脚的功能。

1. V_{dd}(V_{cc})

两个 V_{dd}引脚用来为芯片提供电压，典型的电压源是+5V。为了降低 PIC 系统的噪声和功率损耗，一些 PIC18F 芯片可以使用更低的电压。通过设置配置寄存器的位可以选择其他

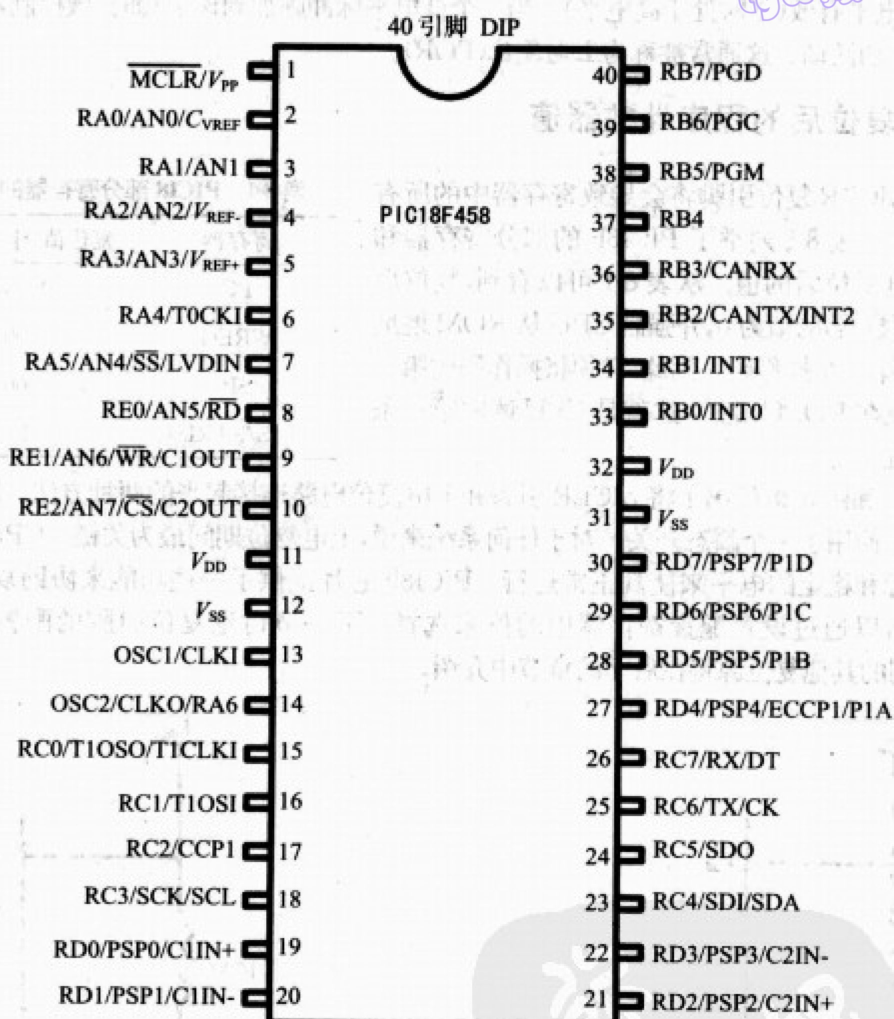


图 8-1 PIC18F458 引脚图

的 V_{dd}电压等级。关于 V_{dd}的配置寄存器将在下一节中讨论。

2. V_{ss}(GND)

这两个 V_{ss}引脚用于接地。在引脚数多于 40 的芯片中,多个 V_{cc}和 GND 引脚是很普遍的,这将有助于在高频系统中降低噪声(接地抖动),详情请参阅附录 C。

3. OSC1 和 OSC2

PIC18F 有许多连接时钟源的方式,最经常使用的是将石英晶体振荡器连接到输入引脚 OSC1 和 OSC2。这样的石英晶体振荡器需要两个电容,每个电容的一端要接地,如图 8-3 所示。注意 PIC18F 微控制器的工作频率范围为 0 Hz~40 MHz。

通过设置配置寄存器的位,可以选择不同的时钟频率。关于振荡器的配置寄存器将在下一节讨论。

4. MCLR

在 PIC18F458 的 40 引脚 DIP 封装中,引脚 1 是 MCLR,即主清零复位引脚。它是一个输

入引脚，低电平有效（平时处于高电平）。当一个低电平脉冲施加到该引脚时，微控制器将复位并且终止一切活动。这通常被称为上电复位(POR)。

301

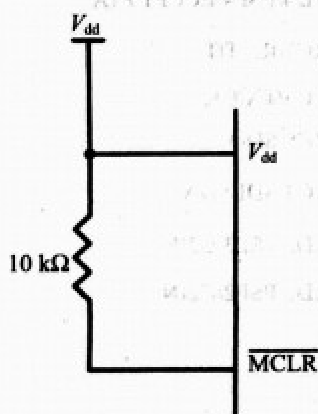
8.1.1 复位后的程序计数器值

使能 MCLR 复位引脚将会导致寄存器中的所有内容都丢失。表 8-1 列举了 PIC18F 的部分寄存器和它们在上电复位后的值。从表 8-1 可以看到，复位后 PC(程序计数器)的值为 0，并强制 CPU 从 ROM 地址 00000 读取第一个操作码。因此，必须将操作码的第一个字节存放在 ROM 地址 0，这就是 CPU 读取第一条指令的位置。

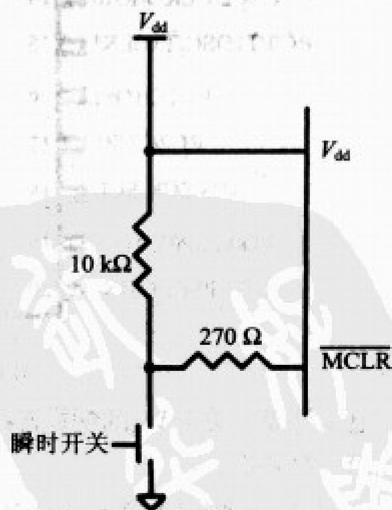
表 8-1 PIC18 部分寄存器的复位值

寄存器	复位值（十六进制）
PC	000000
WREG	00
SP	00
TRISA-TRISE	FF

图 8-2a 和图 8-2b 给出了将 MCLR 引脚和上电复位电路连接起来的两种方法。图8-2b在复位电路中使用了一个瞬态开关。对于任何系统来说，上电复位期间最为关键。CPU 需要稳定的时钟源和稳定的电平来使其正常运行。PIC18F 芯片提供了一些功能来协助复位过程。这些功能可以通过设置配置寄存器中的位来选择。下一节讨论复位引脚的配置寄存器。PIC18 系列的其他复位源将在后面的章节中介绍。



(a) PIC18F458 上电复位电路



(b) PIC18F458 带有手动开关的上电复位电路

图 8-2

在任何 PIC 芯片中，上述的几个引脚都必须连接，即每个 PIC18 都必须有的最小连接。请参阅图 8-3。

正如在第 4 章所看到的，PIC18 系列芯片的端口数目因型号不同而有所不同。下面从另一个方面来研究 PIC18F458 的端口。

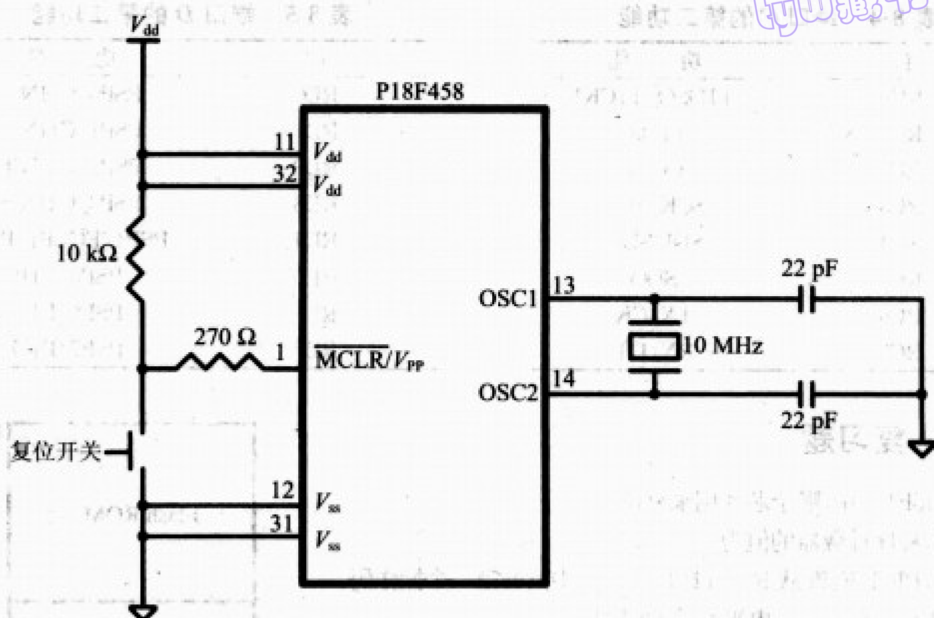


图 8-3 PIC18F458 的最小连接

302

8.1.2 端口 *A*、*B*、*C*、*D* 和 *E*

如图 8-1 所示，端口 *A*、*B*、*C*、*D* 和 *E* 共使用了 33 个引脚，这在第 4 章也讨论过。在复位时所有的端口都被设置成输入，因为 TRISA～TRISE 中的复位值均为 FFH。表 8-2～表 8-5 给出了端口 *A*～*E* 的功能描述以及它们的第二功能。在以后的章节中讨论 PIC18 的功能特征时，将介绍这些引脚的第二功能。

表 8-2 端口 *A*/端口 *E* 的第二功能

位	功 能
RA0	AN0/CVREF
RA1	AN1
RA2	AN2/VREF−
RA3	AN3/VREF+
RA4	T0CKI
RA5	AN4/SS/LVDIN
RA6	OSC2/CLKO
RE0	AN5/RD
RE1	AN6/WR/C1OU
RE2	AN7/CS/C2OUT

表 8-3 端口 *B* 的第二功能

位	功 能
RB0	INT0
RB1	INT1
RB2	INT2/CANTX
RB3	CANRX
RB4	
RB5	PGM
RB6	PGC
RB7	PGD

表 8-4 端口 C 的第二功能

位	功 能
RC0	T1OSO/T1CKI
RC1	T1OSI
RC2	CCP1
RC3	SCK/SCL
RC4	SDI/SDA
RC5	SDO
RC6	TX/CK
RC7	RX/DT

表 8-5 端口 D 的第二功能

位	功 能
RD0	PSP0/C1IN+
RD1	PSP1/C1IN−
RD2	PSP2/C2IN+
RD3	PSP3/C2IN−
RD4	PSP4/ECCP1/P1A
RD5	PSP5/P1B
RD6	PSP6/P1C
RD7	PSP7/P1D

8.1.3 复习题

1. 在 PIC18F458 中，哪个芯片用来复位？
2. 上电后，程序计数器的值为______。
3. 上电后，PIC18F458 从 ROM 地址______中读取第一个操作码。
4. MCLR 是一个______电平有效的引脚。
5. 在 PIC18F458 芯片中，有多少个 V_{dd} 和 GND 引脚？

8.2 PIC18 配置寄存器

通过设定配置寄存器的位能够选择 PIC18 的某些功能特征。通过消除对外部元件的需求，这些功能特征将会降低系统开销。配置寄存器的起始地址为 300000H，如图 8-4 所示。注意，地址 300000H 不在程序 ROM 的 000000～1FFFFFH 地址范围内。在源代码中使用 CONFIG 伪指令可以向配置寄存器写入 8 位数值，每次写入一个字节。换而言之，在应用程序中应给出寄存器的名字和期望的数值，ROM 程序员将它们连同应用程序一起下载到配置寄存器。用户程序可以通过表读取和表写入指令来访问配置寄存器。本节将介绍一些基本的配置寄存器，如复位、时钟源和 V_{dd} 电压。Microchip 网站提供了 PIC 微控制器配置寄存器的完整列表。如果想了解某个型号 PIC18 的配置寄存器，可以在 Microchip 网站查阅 Configuration Register Settings Addendum（配置寄存器设置补充）文档。表 8-6 简单地描述了配置寄存器。必须要注意的是，如果一个配置寄存器没有被正确地编程，它将会导致整个系统的失败。其中一个例子就是改变连接到微控制器的时钟源类型。

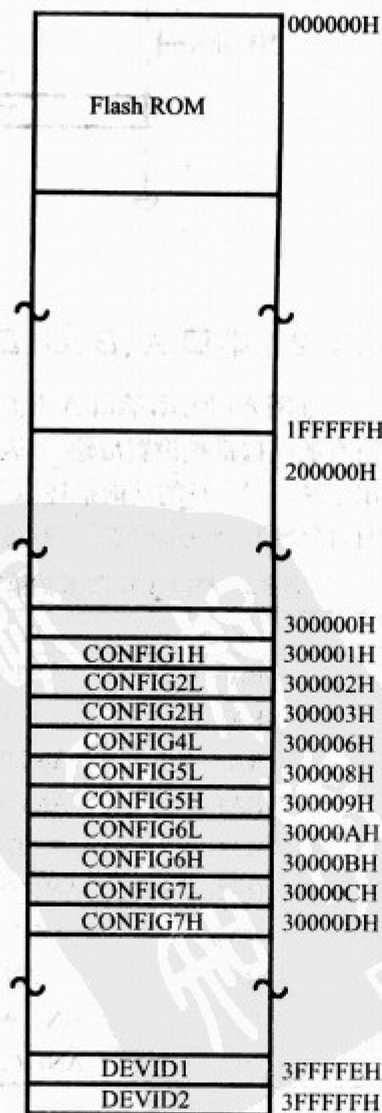


图 8-4 CONFIG 寄存器的存储空间映射图

表 8-6 PIC18F458 的配置寄存器

地址(十六进制)	名　字	通用描述
300001	CONFIG1H	振荡器选择
300002	CONFIG2L	掉电
300003	CONFIG2H	看门狗启用
300006	CONFIG4L	背景调试模式和 ISCP
300008	CONFIG5L	代码保护
300009	CONFIG5H	EEPROM 和启动空间保护
30000A	CONFIG6L	写保护
30000B	CONFIG6H	写保护
30000C	CONFIG7L	读保护
30000D	CONFIG7H	引导程序块读保护
3FFFFE	DEVID1	设备 ID 和修正
3FFFFF	DEVID2	设备 ID

304

8.2.1 CONFIG1H 寄存器和振荡器时钟源

CONFIG1H 寄存器地址为 0x300001,用于配置时钟振荡器,如图 8-5 所示。下面是对 CONFIG1H 寄存器位选项的描述。

1. FOSC2～FOSC0

FOSC2、FOSC1 和 FOSC0 是用来选择 CPU 的时钟频率的。默认的选择是 RC(111),即选择使用连接有外部电阻和电容的片上振荡器。在该选择下,设计员要做的就是将 OSC1 引脚同 RC 电路连接起来。时钟速度由电阻和电容的数值决定。采用这种方式向 CPU 提供时钟源时,OSC2(PORTA 的第 6 位)可以用作 I/O 引脚。如果使用选项 101(EC:外部时钟),并向 OSC1 引脚提供一个外部时钟源,那么 RA6 可用作 I/O 引脚。这种设置与使用选项 100 相同的是,OSC2 能提供一个 OSC/4 的频率。OSC/4 的时钟可以用来让所有的系统活动与 CPU 同步。最广泛应用的方法是将 OSC1 和 OSC2 两个引脚连接到一个晶体(或陶瓷)振荡器,如图 8-6 所示。对于晶体振荡器,有 4 种选择,分别是 PPLHS、HS、XT 和 LP。它们的主要区别是频率范围不同,如表 8-7 所示。使用 LP(低功率)时功率损耗最低,而使用 PPLHS(锁相环高速)则功率损耗最高。注意,如附录 C 所介绍的,频率越高,CPU 消耗的功率越大。在本书中,许多电路都使用 HS(高速)项。如果将 OSC1—OSC2 连接到一个 10 MHz 的晶体振荡器并选择 PPLHS,那么 CPU 会工作在 40 MHz,这是因为 PPLHS 利用锁相环技术将提供给 CPU 的时钟频率提高了 4 倍。PLLHS 的功率消耗也是最高的。注意,选择 RC(111)是最经济的,而选择 LP(000)则功耗最低。

U-0	U-0	R/P-1	U-0	U-0	R/P-1	R/P-1	R/P-1
—	—	$\overline{OSCSEN}$	—	—	FOSC2	FOSC1	FOSC0

第7位　　　　　　　　　　　　　　　　　　　　　　　　第0位

位7、位6 未实现：读为0

位5　$\overline{OSCSEN}$：晶体振荡器时钟开关启用位

1=禁止晶体振荡器时钟开关（主振荡器是时钟源）

2=启用晶体振荡器时钟开关（振荡器开关启用）

位4、位3 未实现：读为0

位 2~0 FOSC2：FSC0：振荡器选择位

111=RC振荡器，OSC2被配置为RA6

110=HS振荡器，带有PLL使能端，时钟频率=4×F_{OSC}

101=EC振荡器，OSC2被配置为RA6

100=EC振荡器，OSC2被配置为1/4时钟频率输出

011=RC振荡器

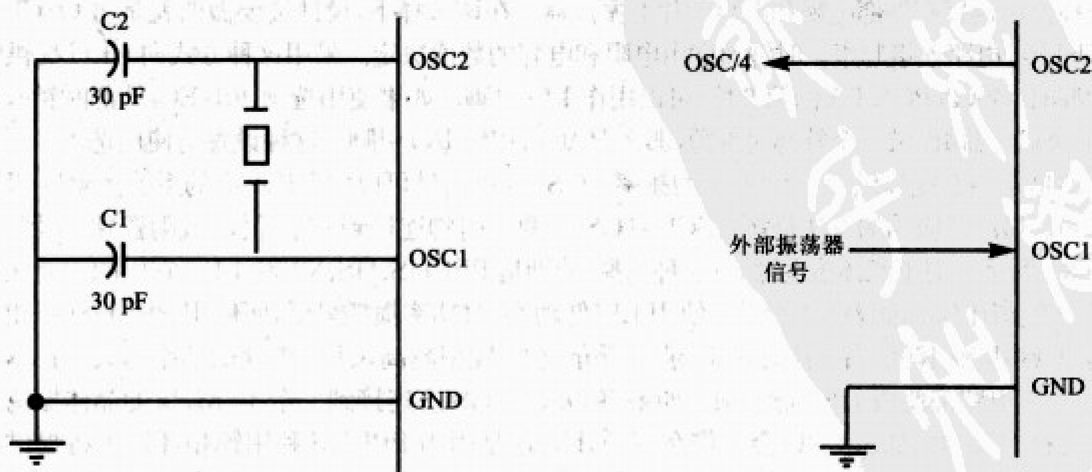
010=HS振荡器

001=XT振荡器

000=LP振荡器

图例说明：		
R=可读位	P=可编程位	U=未实现位，读为0
-n=当设备未被编程时的值	u=在编程状态时无变化	

图 8-5　CONFIG1H 寄存器用于频率选择



(a) OSC1 和 OSC2 与晶体振荡器的连接　　(b) OSC 与外部时钟源的连接

图 8-6　OSC 连接

2. OSCSEN

CONFIG1H 的 OSCSEN 位(D5)允许 CPU 切换到一个内部时钟源，这个内部时钟源有

32 kHz 的固定频率。将时钟源从连接 OSC1 和 OSC2 的外部振荡器切换到一个 32 kHz 的内部时钟源，在许多电池供电的系统中可以将功率消耗降到最低。使用该选项并将晶体频率选择为 LP 模式，可以将 CPU 的功率消耗降低到纳瓦级。注意，32 kHz 的低频时钟源不同于连接至 OSC1 和 OSC2 引脚的外部时钟源。该 32 kHz 的辅助时钟源独立于 OSC1—OSC2 时钟源，而且在晶体频率信号失败的情况下可以继续为 CPU 提供时钟。本书禁用辅助时钟源，使用 OSC1—OSC2 振荡器作为主要的时钟源。

表 8-7 PIC18F458 振荡器频率选择及电容值范围

OSC 选择	晶振频率	C1 取值范围	C2 取值范围
LP	32 kHz	33 pF	33 pF
LP	200 kHz	15 pF	15 pF
XT	200 kHz	47～65 pF	47～65 pF
XT	1 MHz	15 pF	15 pF
XT	4 MHz	15 pF	15 pF
HS	4 MHz	15 pF	15 pF
HS	8 MHz	15～33 pF	15～33 pF
HS	20 MHz	15～33 pF	15～33 pF
HS	25 MHz	15～33 pF	15～33 pF

306

3. 振荡器频率和指令时钟周期

第 2 章～第 4 章已经介绍过指令周期时间以及如何设计时延子例程。在 PIC18 微控制器中，指令周期时间是以提供给 OSC 引脚的时钟源的 1/4 为基础的。这将在例8-1中被再次讨论。

例 8-1 对于下面连接到 OSC1 和 OSC2 引脚的晶体振荡器，计算出 PIC18F458 芯片的指令周期时间。对于每种晶振频率，CONFIG1H 的模式选择如下：

(a) 4 MHz，XT (b) 10 MHz，HS (c) 20 MHz，HS

解：所有的 CONFIG1H 的选项都使用 1/4 时钟源频率来作为指令周期时间。

(a) 由于 4 MHz/4=1 MHz，所以指令周期时间为 1/1 MHz=1 μs。

(b) 由于 10 MHz/4=2.5 MHz，所以指令周期时间为 1/2.5 MHz=0.4 μs=400 ns。

(c) 由于 20 MHz/4=5 MHz，所以指令周期时间为 1/5 MHz=0.2 μs=200 ns。

如果使用 10 MHz 的晶体振荡器并且选择 HSPLL 模式(代替 HS)，这时 CPU 的时钟源频率为 40 MHz。这意味着指令周期时间是 1/10 MHz=0.1 μs=100 ns，因为 40 MHz/4=10 MHz。

307

4. CONFIG 指令

表 8-8 给出了 MPLAB 支持的 CONFIG1H 字节的语法选择。每当将应用程序下载到 PIC18 程序 ROM 时，需要将 CONFIG 字节数也加载到配置寄存器中。在源程序中使用 CONFIG 指令可以完成这个任务。在源代码中，根据表 8-8，使用 CONFIG 指令设置 CONFIG1H 的值，如下所示：

```
CONFIG OSC = HS              ;high-speed oscillator
CONFIG OSCS = OFF            ;disable Osc switch
```

或者，将它们合并成一条指令，如下所示：

```
CONFIG OSC = HS, OSCS = OFF  ;oscillator, no Osc switch
```

表 8-8　在 MPLAB 中使用 CONFIG 伪指令的 CONFIGIH 选项

晶体振荡器选择		
OSC = LP	LP	低功耗
OSC = XT	XT	晶体振荡器
OSC = HS	HS	高速
OSC = RC	RC	电阻/电容
OSC = EC	EC 、OSC2 作为 Clock Out	外部时钟
OSC = ECIO	EC、OSC2 作为 RA6	外部时钟
OSC = HSPLL	HS-PLL 启用	高速锁相环
OSC = RCIO	EC、OSC2 作为 RA6	外部时钟
晶体振荡器使能开关		
OSCS = ON	启用	
OSCS = OFF	禁用	

8.2.2　CONFIG2L 寄存器和复位电压

CONFIG2L 的地址是 0x300002，用于在芯片复位期间提供稳定的电压和时钟频率，如图 8-7 所示。对于一个系统来说，上电复位期间最为关键。CPU 需要稳定的时钟源和稳定的电平来使其正常地运行。有助于实现这个目标的两个内部定时器，它们是上电定时器(PWRT)和振荡器启动定时器(OST)。这两个内部定时器有助于降低上电过程中由频率和电压源引起的延时。PWRT 提供了一个固定时延时间，它将保持 CPU 的复位状态直到电源稳定。OST 定时器对于晶振频率也有同样的作用。这两个定时器使得芯片没有必要再在上电过程中使用用于电压和频率稳定的外部电路。CONFIG2L 允许人们设置电压和频率来保持 CPU 的复位状态，直到时钟和电压都变得稳定为止。接下来讨论这个重要的配置寄存器的每个位的设置。

308

1. BORV1：BORV0

供给 V_{cc}(V_{dd})引脚的电源波动偶尔会引起 CPU 工作不正常。为此，PIC18 系列芯片提供一个掉电复位电压。CONFIG2L 的掉电复位电压位用于设置 V_{dd}的最小电压值。一旦 V_{dd}的电压低于这个最小值，CPU 将进入复位状态并终止所有工作。这样的设置是必要的，因为 V_{dd}(V_{cc})引脚电压是根据连接到 OSC1 和 OCS2 引脚的晶振频率来设置的。在频率为 40 MHz、V_{dd}=5 V 时，可以将 BORV 设为 4.5 V。如果 V_{dd}低于 4.5 V，CPU 将进入复位状态，停止任何程序的执行，并保留寄存器中的数据。对于一个频率为 2 MHz 或者更低的低功率系统，可以将 V_{dd}

U-0	U-0	U-0	U-0	R/P-1	R/P-1	R/P-1	R/P-1
—	—	—	—	BORV1	BORV0	BOREN	PWRTEN

第7位　　　　　　　　　　　　　　　　　　第0位

位7~4　未实现：读为0

位3~2　BORV1：BORV0：掉电复位电压位

11=VBOR被设为2.0 V

10=VBOR被设为2.7 V

01=VBOR被设为4.2 V

00=VBOR被设为4.5 V

位1　BOREN：掉电复位使能位

1= 启用掉电复位

0= 禁用掉电复位

位0　PWRTEN：上电定时器使能位

1=禁用PWRT

0=启用PWRT

图例说明：

R=可读位　　P=可编程位　　U=未实现位，读为0

-n=当设备未被编程时的值　　u=在编程状态时无变化

图 8-7　用于复位电压的 CONFIG2L 配置寄存器

设为 2 V,相应地设 BOVR=1.8 V。同样,当 V_{dd}低于 1.8 V时,CPU 会进入复位状态。当 V_{dd}大于 1.8V 时,CPU 又会跳出复位状态并继续执行程序。因此,CONFIG2L 的 BOVR1:BORV0 位要根据供给 V_{dd}引脚的 V_{dd}的电压和连接到 OSC0 和 OSC1 的振荡器频率来确定。在本书中,将 BORV 设置为 4.5 V,因为 V_{dd}=5 V,且晶体振荡器频率为 10 MHz。

2. BOREN

该位用来启用上述的 BORV1：BORV0 位。

3. PWRTEN

该位用来启用上电定时器(PWRT)。PWRT 在上电期间提供固定的延迟时间,用以保持 CPU 为复位状态,直到电压稳定。

表 8-9 给出了 MPLAB 支持的提供了 CONFIG2L 的语法选择。在源代码中,根据表 8-9,使用 CONFIG 指令设置 CONFIG2H 的值,如下所示:

```
CONFIG BORV=45          ;for Vdd = 5 V, OSC = 10 MHz
CONFIG PWRT = ON        ;use power-timer
CONFIG BOR=ON           ;enable BORV option
```

或者,将它们合并成一条指令,如下所示:

```
CONFIG BORV = 45, PWRT = ON,  BOR=ON
```

表 8-9 PIC18F458 的 CONFIG2L 寄存器选项

掉电电压		上电定时器	
BORV = 45	4.5 V	PWRT = ON	启用
BORV = 42	4.2 V	PWRT = OFF	禁用
		掉电复位	
BORV = 27	2.7 V	BOR = ON	启用
BORV = 20	2.0 V	BOR = OFF	禁用

8.2.3 CONFIG2H 寄存器和看门狗定时器

CONFIG2H 寄存器的地址是 0x300003，用于看门狗定时器。近几年的微控制器都带有一个看门狗定时器。当系统由于代码执行出错而挂起或者失控时，可以使用看门狗定时器强制微控制器进入已知的复位状态。在嵌入式系统中，看门狗定时器有很多应用。其中一个应用就是使用看门狗定时器来阻止系统因软件错误而进入死循环。看门狗的另一个应用是捕捉导致系统挂起的事件。这些问题常发生在程序 ROM 遭到破坏时，主要是由电源浪涌、电气噪声环境或者程序计数器值的突然改变等导致的。在这些情况下，看门狗定时器将强制系统

310

U-0	U-0	U-0	U-0	R/P-1	R/P-1	R/P-1	R/P-1
—	—	—	—	WDTPS2	WDTPS1	WDTPS0	WDTEN

第7位 第0位

位7~4 未实现：读为0

位3~1 WDTPS2：WDTPS0：看门狗前分频器选择位
111=1:128
110=1:64
101=1:32
100=1:16
011=1:8
010=1:4
001=1:2
000=1:1

注意：用于PIC18Fxxx设备的看门狗定时器前分频器选择位的配置是由PIC18Cxxx设备中的配置变化而来的。

位0 WDTEN：看门狗定时器使能位
1=启用WDT
0=禁用WDT（对SWDTEN位施加控制）

图例说明：
R=可读位 P=可编程位 U=未实现位，读为0
-n=当设备未被编程时的值 u=在编程状态时无变化

图 8-8 CONFIG2H 配置寄存器用于看门狗定时器

从要恢复的状态进入已知的复位状态。在一些应用中，系统在没有工作时将进入睡眠状态，以节约电池供电。这时，看门狗定时器可以用来监测键盘，当键盘上有操作时，立即唤醒系统处理信息。图 8-8 是对 CONFIG2H 寄存器的说明。

1. WDTEN

该位启用看门狗定时器。

2. WDTPS2：WDTPS0

看门狗定时器的前分频器选择位允许将 WDT 最多设为 2 分钟。附录 A 讨论了设置带 WDT 的 SLEEP 指令。

对于本书的应用程序，我们关闭了看门狗定时器。可以在 MPLAB 中关闭看门狗定时器或者在源代码中根据表 8-10 使用 CONFIG 指令来设置 CONFIG2H 的值，如下所示：

```
CONFIG WDT = OFF
```

表 8-10 PIC18F458 的 CONFIG2H 寄存器选项

看门狗定时器	
WDT = ON	启用
WDT = OFF	禁用

8.2.4 CONFIG4L 寄存器和背景调试程序

CONFIG4L 地址是 0x300006，用于启用背景调试程序，如图 8-9 所示。表 8-11 给出了 CONFIG4L 的各个选项，下面将逐一介绍。

R/P-1	U-0	U-0	U-0	U-0	R/P-1	U-0	R/P-1
$\overline{\text{DEBUG}}$	—	—	—	—	LVP	—	STVREN
第7位							第0位

位7　$\overline{\text{DEBUG}}$：背景调试器使能位

1=禁用背景调试器。RB6和RB7配置为通用I/O引脚

0=启用背景调试器。RB6和RB7专用于电路内调试

位6~3　未实现：读为0

位2　LVP：低电压ICSP使能位

1= 启用低电压ICSP

0= 禁用低电压ICSP

位1　未实现：读为0

位0　STVREN：栈满/下溢出复位使能位

1=栈满/下溢出将引起复位

0=栈满/下溢出不引起复位

图例说明：

R=可读位	C=清零位	U=未实现位，读为0
-n=当设备未被编程时的值	u=在编程状态时无变化	

图 8-9　CONFIG4L 配置寄存器用于背景调试器

1. DEBUG

如果将 PIC18 连接到一个电路内部调试器，那么就不能使用 PORTB 的引脚 RB6 和 RB7。如果禁止 CONFIG4L 字节的背景调试器选项，可以将 RB6 和 RB7 用作普通的 I/O 引脚。

2. STVREN

D0 位用于栈溢出。正如第 3 章所讨论的，PIC18 只有 31 个地址来作为栈区。在启用 D0 位时，如果栈溢出或者下溢出，可以让系统进入复位状态。

3. LVP

D2 位用于启用引脚 RB5 的低电压电路内串行编程（ICSP）。同样，可以禁止 D2 位，将 RB5 用作 I/O 引脚。

311

表 8-11 PIC18F458 的 CONFIG4L 寄存器选项

背景调试器使能	
DEBUG = ON	启用
DEBUG = OFF	禁用
低电压 ICSP	
LVP = ON	启用
LVP = OFF	禁用
栈溢出复位	
STVR = ON	启用
STVR = OFF	禁用

表 8-11 说明了 MPLAB 仿真器使用的 CONFIG4L 字节的选择语法。对于本书的应用程序，使用 CONFIG 伪指令关闭调试器、LVP 和栈溢出的所有选项，指令如下：

```
CONFIG DEBUG = OFF, LVP = OFF, STVR = OFF
```

以上 4 个 CONFIG 寄存器是任何 PIC18F452 或基于 458 的系统所必需的最小数量寄存器。其余的寄存器专用于程序和数据保护。具体信息请参阅 Microchip 的网站。

设置配置寄存器的注意事项

（1）PIC18 系列的每个型号都有自己的配置寄存器值。其取值可参考 Microchip 公司网站。

（2）Microchip 公司建议在 PIC18 系列应用中使用 CONFIG 伪指令代替 _CONFIG。虽然_CONFIG 伪指令（注意_CONFIG 有两条下划线）能运行在 PIC18 上，但是不建议使用它。根据 Microchip 公司的建议，不可以在同一程序中同时使用 CONFIG 伪指令和_CONFIG 伪指令。

312

8.2.5 LIST 伪指令

LIST 伪指令是在源程序中使用的又一条伪指令，用于要烧录到 PIC ROM 的程序。LIST 伪指令告知 MPLAB 汇编程序一些选项设置，如 Intel 十六进制文件格式、基本数据格式、源代码打印输出等。表 8-12 给出了本书中用到的 LIST 伪指令的主要选项。

下面是如何使用 LIST 伪指令的一个例子：

```
LIST  P=18F458, F=INTHX32, MM=OFF, R=HEX, ST=OFF X=OFF
```

表 8-12 中的一些选项可以由 MPLAB 汇编程序自行设定。为了保证在共享源文件时选项已设置好，可以使用 LIST 指令来设置它们。

表 8-12 一些 LIST 伪指令选项

B=nnn	设置表格间距(默认值为 8)
C=nnn	设置打印输出的列宽(默认值为 132)
F=格式	设置十六进制文件输出。可以选择 INHX32、INHX8M 或者 INHX8S。默认值为 INHX8M(详见下节的相关内容)
MM={ON/OFF}	在列表文件中打印存储映像表(默认值为 ON)
N=nnn	设置打印输出的每页行数(默认值是 60)
P=type	设置微控制器的类型(如 P=PIC18F458)
R=radix	设置源代码数据格式。选项为十六进制、十进制和八进制(默认值为十六进制)
ST={ON/OFF}	打印列表文件中的符号表(默认为打开)
X={ON/OFF}	打开或者关闭宏扩展(默认为打开)

8.2.6 设置所有的配置寄存器

前 7 章给出的所有程序都是可以仿真的。然而,为了创建可烧录的程序,必须在汇编和连接程序之前给出所有的配置寄存器字节值,并在源代码中设置 LIST 伪指令的期望选项。这样,由 MPLAB 汇编的十六进制输出文件就可以用 ROM 烧录器写入 PIC18 芯片程序 ROM 中。由于所有的配置寄存器都已经设置完毕,还可以将这样的十六进制文件发送给其他用户使用。对于将要烧录到 ROM 中的程序,可以使用下面的源代码:

313

```
;skeleton of a PIC18 Assembly language program
        LIST  P=PIC18F458, F=INHX32, MM=OFF,  N=0, ST=OFF, R=HEX
        #include P18F458.INC
        CONFIG OSC=HS, OSCS=OFF          ;high-speed XTAL as clk src
        CONFIG WDT=OFF                   ;disable watchdog timer

        ;Brown-out Reset Volt = 4.5 V and Power-up Timer is on
        CONFIG BORV=45, PWRT=ON, BOR=ON

        ;no Background debug, no Reset if stack overflows
        ;and pin PB5 = I/O
        CONFIG DEBUG=OFF, LVP=OFF, STVR=OFF
        .......
        ORG 0
        .......
        .......
        .......
        END
```

作为例子,请分析下面的程序,它将 PORTB 的所有位有延时间隔地在开和关之间翻转。

```
;Test Program 8-1: Toggling PORTB for the PIC18F458 and
;XTAL = 10 MHz
        LIST P=PIC18F458, F=INHX32, N=0, ST=OFF, R=HEX
        #include P18F458.INC
        CONFIG OSC = HS, OSCS = OFF
        CONFIG WDT = OFF
        CONFIG BORV = 45, PWRT = ON, BOR = ON
        CONFIG DEBUG = OFF, LVP = OFF, STVR = OFF
```

```
R1      EQU     0x07
R2      EQU     0x08
R3      EQU     0x09
        ORG 0
        CLRF    TRISB           ;make Port B an output port
        MOVLW   0x55            ;WREG = 55h
        MOVWF   PORTB           ;put 55h on port B pins
L3      COMF    PORTB,F         ;toggle bits of Port B
        CALL    QDELAY          ;quarter of a second delay
        BRA     L3              ;continue

;-----------1/4 SECOND DELAY
QDELAY
        MOVLW D'2'
        MOVWF R1
D1      MOVLW D'250'
        MOVWF R2
D2      MOVLW D'250'
        MOVWF R3
D3      NOP
        NOP
        DECF    R3, F
        BNZ     D3
        DECF    R2, F
        BNZ     D2
        DECF    R1, F
        BNZ     D1
        RETURN
        END
```

314

8.2.7 在 MPLAB C18 C 编译器中设置 CONFIG 寄存器

在第 7 章讨论了如何使用 C18 C 编译器的 PIC18F 的 C 语言编程。那些程序是可以仿真的。为了创建可烧录的 C 程序，必须保证对配置寄存器的设置。一种方法是使用 #pragma 指令。下面的 C18 程序框架可用于创建烧录到 ROM 的程序。

```
;skeleton of a PIC18 C18 C language program
        #pragma config OSC = HS, OSCS = OFF
        #pragma config BORV = 45, PWRT = ON, BOR = ON
        #pragma config WDT = OFF
        #pragma config DEBUG = OFF, LVP = OFF, STVR = OFF

        void main (void)
        {
        ......
        ......
        ......
        }
```

作为例子，请分析下面的程序，它将 PORTB 的所有位有延时间隔地在开和关之间翻转。

```
;Test Program 8-2: Toggling PORTB for the PIC18F458 and
;XTAL = 10 MHz
        #pragma config OSC = HS, OSCS = OFF
        #pragma config PWRT = OFF, BOR = ON, BORV = 45
        #pragma config WDT = OFF
        #pragma config DEBUG = OFF, LVP = OFF , STVR = OFF
        #include <P18F458.h>

        void msdelay(unsigned int ms);
```

```
void main(void)
  {
    TRISB = 0;                    //make Port B an output
    while(1)
      {
      PORTB = 0x55;
      msdelay(500);
      PORTB = 0xAA;
      msdelay(500);
      }
  }

//this delay is for a 10 MHz clock
void msdelay(unsigned int ms)
  {
  unsigned int x;
  unsigned char z;
  for(x=0;x<ms;x++)
    for(z=0;z<165;z++);
  }
```

315

8.2.8 复习题

1. 一个给定的 PIC18F458 系统有一个 16 MHz 的晶体振荡频率，CONFIG1H 设置为 HS 状态。请确定 CPU 的指令周期时间。
2. CONFIG1H 寄存器的地址是多少?
3. 判断对错:上电后,电压和频率都立即稳定。
4. 用于 OSC1—OSC2 频率的 LP 选项的工作频率范围是______到______kHz。
5. 哪个配置寄存器是用来禁用看门狗定时器的? 地址是多少?
6. 判断对错:上电后,上电定时器让 CPU 一直处于复位状态直到电压稳定。
7. 判断对错:配置寄存器地址处于 ROM 程序地址空间内。
8. 判断对错:对于一个低频系统,掉电复位电压可以被设置成一个较低的电压。
9. 判断对错:系统的时钟频率越高,功率消耗越低。
10. 如果在源代码中定义 BORV=42,那么在 V_{dd} 电压低于多少时 CPU 会进入复位状态?

8.3 解释 PIC18 的 Intel 十六进制文件

Intel 十六进制文件是一种广泛应用的文件格式,它使得可执行机器代码下载到 ROM 芯片的过程标准化。因此,用于 ROM 烧录器(编程器)的下载程序支持 Intel 十六进制文件格式。在很多基于 Windows 的汇编工具(如 MPLAB)中,Intel 十六进制文件是根据你的设置而生成的。在 PIC MPLAB 环境中,将目标文件输入到连接程序,用以产生 Intel 十六进制文件。EPROM 编程器(如 PICkit 2 编程器)的下载程序将十六进制文件下载(传输)到 ROM 中。MPLAB 汇编程序可以生成 3 种类型的 Intel 十六进制文件,分别是 INHX8M、INHX32 和 INHX8S,如表 8-13 所示。本节将通过一些例子来深入分析每一种文件类型。

表8-13 由MPLAB生成的Intel十六进制文件格式(请参阅 http://www.microchip.com)

格式名称	文件格式	文件扩展名	最大ROM地址
Intel 十六进制格式	INHX8M	.hex	16位地址
Intel 十六进制32格式	INHX32	.hex	32位地址
Intel 分段十六进制	INHX8S	.hxl 和.hxh	每段都是16位地址

8.3.1 分析Intel十六进制(INHX8M)文件

通过使用LIST伪指令或者在MPLAB汇编工具中设定选项来选择Intel十六进制文件类型(INHX8M、INHX32或INHX8S)。如果没有选择,那么MPLAB汇编器将默认选择INHX32类型。接下来,对于INHX8M,分析属于列表文件的十六进制文件。INHX8M文件是MPLAB汇编工具通过选择LIST伪指令中的INHX8M选项(或者设置MPLAB工具)而生成的。文件扩展名为.hex。INHX8M用于空间不大于64 KB的程序ROM。为了创建适用于大于64 KB的程序ROM的Intel十六进制文件,必须使用INHX32选项。图8-10给出了测试程序的INHX8M十六进制文件,其列表文件已在前面给出。注意在LIST伪指令中选择INHX8M选项。因为ROM烧录器(下载程序)使用的是十六进制文件将操作码下载到ROM中,所以十六进制文件应该包括下载信息的字节数、信息体和信息存放的起始地址。十六进制文件的每行包含以下的6个部分:

:BBAAAATTHHHHH……HHHHCC

下面是对各部分的说明。

```
:10000000936A550E816E811E07EC00F0FCD7020E3C
:10001000076EFA0E086EFA0E096E0000000009065F
:0C002000FCE10806F8E10706F4E112001C
:0300010022020ECA
:010006008079
:060008000FC00FE00F40E5
:00000001FF

Separating the fields, we get the following:

:BB AAAA TT HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH        CC
:10 0000 00 936A550E816E811E07EC00F0FCD7020E        3C
:10 0010 00 076EFA0E086EFA0E096E000000000906        5F
:0C 0020 00 FCE10806F8E10706F4E11200                1C

:03 0001 00 22020E                                  CA
:01 0006 00 80                                      79
:06 0008 00 0FC00FE00F40                            E5
:00 0000 01                                         FF
```

图8-10 带有INHX8M选项的Intel十六进制文件测试程序

(1) “:”每行以冒号开始。

(2) BB,字节数计数值。它告诉下载程序本行有多少字节要下载。BB 的范围可以是 00～16(即十六进制数 10)。

(3) AAAA 表示地址。这是用于 INHX8M 的一个十六位地址。下载程序把数据的第一个字节放到这个存储地址中。

(4) TT 表示类型。TT 的值可以为 00 或者为 01。如果值是 00,表明在这一行后面有更多的行。如果值是 01,说明这是最后一行,在它之后下载结束。

(5) HH…H 代表实际的信息(数据或代码)。这部分最大为 16 B。下载程序将这些信息放入一个连续的 ROM 空间中。因为 PIC18 拥有 16 位带宽的程序 ROM 空间,所以信息的存放顺序是先存低位后存高位。

(6) CC 是一个独立的字节。这最后的单字节是该行所有信息的校验和。校验和字节数用于错误检测。关于校验和字节数已在第 6 章和第 7 章中讨论过。注意,每行末尾的校验和字节数是该行所有信息的校验和,而不仅仅是数据部分的校验和。

现在,将图 8-10 中 Intel 十六进制文件的数据部分和图 8-11 中 .lst 文件的 OBJ 字段下的信息进行比较。注意,它们是一致的。Intel 十六进制文件格式增加了额外的信息。读者可以运行测试程序的 C 语言版本,并验证其操作。如果读者想研究 Intel 十六进制文件的概念,C 语言编译器可以同时提供 .lst 文件和 Intel 十六进制文件。

例 8-2 分析图 8-10 中第 3 行的 6 个部分。

解:在冒号后面,可以得到 0C,它说明这一行有 12B 的数据。0020 是数据的起始地址。接下来,00 表示这不是最后一行。FCE1080 6F8E 10706F4E 11200 是 12B 的数据。最后一个字节 1C 是校验和。

例 8-3 比较图 8-10 中 Intel 十六进制文件的数据部分和图 8-11 中测试程序的列表文件中的操作码,观察它们是否一致。

解:在图 8-10 的第一行中,数据部分以 936AH 开始。因为存放数据的顺序是先低位字节后高位字节,所以指令 CLR FTRISB 的操作码应该是 6A93,正如图 8-11 列表文件所示。第三行中数据部分的最后一个字节是 1200,它是列表文件中 RETURN 指令的操作码。

例 8-4 验证图 8-10 中第三行的检验和,同时验证信息是否出错。

解:0C+ 20+ FC+ E1+ 08+ 06+ F8+ E1+ 07+ 06+ F4+ E1+ 12+ 00= 5E4H(十六进制数)。将最高位的 5 舍去,得到 E4H,它的 2 进制补码是 1CH,这刚好等于第 3 行的最后一个字节。

如果将第 4 行所有的信息加起来(包括校验和字节数),去掉进位后将得到 0C+ 20+ FC+ E1+ 08+ 06 + F8+ E1+ 07+ 06+ F4+ E1+ 12+ 00= 600H。

```
LOC    OBJ              LINE
                        00003 LIST P=PIC18F458,F=INHX8M,N=0,ST=OFF,R=HEX
                        00004 #include P18F458.INC

22 02 0E 80 0F          00005         CONFIG  OSC=HS, OSCS=OFF
C0 0F E0 0F 40
                        00006         CONFIG  BORV=45,PWRT=ON, BOR=ON
                        00007         CONFIG  WDT=OFF
                        00008         CONFIG  DEBUG=OFF, LVP=OFF, STVR=OFF
                        00009

00000007                00010 R1 EQU 0x07
00000008                00011 R2 EQU 0x08
00000009                00012 R3 EQU 0x09

000000                  00014         ORG 0
000000 6A93             00015         CLRF   TRISB
000002 0E55             00016         MOVLW  0x55
000004 6E81             00017         MOVWF  PORTB
000006 1E81             00018 L3      COMF   PORTB,F
000008 EC07 F000        00019         CALL   QDELAY
00000C D7FC             00020         BRA    L3

                        00023 ;-----------1/4 SECOND DELAY
00000E                  00024 QDELAY
00000E 0E02             00025         MOVLW D'2'
000010 6E07             00026         MOVWF R1
000012 0EFA             00027 D1      MOVLW D'250'
000014 6E08             00028         MOVWF R2
000016 0EFA             00029 D2      MOVLW D'250'
000018 6E09             00030         MOVWF R3
00001A 0000             00031 D3      NOP
00001C 0000             00032         NOP
00001E 0609             00033         DECF   R3, F
000020 E1FC             00034         BNZ    D3
000022 0608             00035         DECF   R2, F
000024 E1F8             00036         BNZ    D2
000026 0607             00037         DECF   R1, F
000028 E1F4             00038         BNZ    D1
00002A 0012             00039         RETURN
                        00040         END
```

318 ~ 319

图 8-11　带有 INHX8M 选项的测试程序的列表文件(为简化空间,注释和其他行已删除)

8.3.2　分析 Intel 十六进制文件(INHX32)

对于程序 ROM 空间大于 64 KB 的 PIC 芯片,必须选择 INHX 32 选项。图 8-13 给出了使用 INHX 32 代替 INHX8M 后的测试程序(如图 8-14 所示)的 Intel 十六进制文件。注意,INHX8M 是用于 ROM 空间不大于 64 KB 的芯片,而对于 ROM 大于 64 KB 的芯片则使用

INHX32。INHX32 文件地址是 32 位的，所以 INHX32 能用于地址范围为 1 B～4 GB 的芯片，因为 2^{32}=4 GB，如图 8-12 所示。

注意，跟 INHX8M 选项一样，MPLAB 生成具有 INHX32 类型的扩展名为 .hex 的文件。INHX32 与 INHX8M 非常相似，只是在 TT 字段有额外的选项来驻留 ROM 芯片的 32 位地址。和 INHX8M 类似，INHX32 文件的每一行都包括 6 部分：

:BBAAAATTHHHHH……HHHHCC

00000000H
4 GB ROM空间
FFFFFFFFH

图 8-12　32 位地址芯片的 ROM 空间

```
:020000040000FA
:0E000000936A550E816E811E78EC94F0FCD749
:020000040001F9
:1028F000020E076EFA0E086EFA0E096E0000000056
:0E2900000906FCE10806F8E10706F4E1120002
:020000040030CA
:0600010022020E830180C3
:06000800FFC0FFE0FF4015
:00000001FF

Separating the fields we get the following:

:BB AAAA TT HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH     CC
:02 0000 04 0000                                 FA
:0E 0000 00 936A550E816E811E78EC94F0FCD7         49
:02 0000 04 0001                                 F9
:10 28F0 00 020E076EFA0E086EFA0E096E00000000     56
:0E 2900 00 0906FCE10806F8E10706F4E11200         02
:02 0000 04 0030                                 CA
:06 0001 00 22020E830180                         C3
:06 0008 00 FFC0FFE0FF40                         15
:00 0000 01                                      FF
```

图 8-13　汇编器提供的 Intel 十六进制文件（INHX32）的测试程序

下面是对每个部分的说明。

(1) “:”每一行都以一个冒号开头。

(2) BB，字节数计数值。它告诉下载程序本行有多少字节要下载。BB 的范围可以是 00～16（即十六进制数 10）。

(3) AAAA 是一个 A15～A0 的十六位地址，或者全零。它的值取决于 TT 的状态。

(4) TT 是记录类型。这个字段的值有如下 4 种可能：

00＝数据记录在字段 HHHH 中

01＝文件结束。意味着这是最后一行，在它之后下载结束

02＝段地址记录

04＝线性地址记录在 HHHH 字段（32 位地址的 A31～A16 由 HHHH 字段给出）。注意到如果＝04，那么 HHHH 的值是高 16 位地址 A31～A16，而 AAAA＝0000

(5) HH……H 是实际的信息（数据、地址或代码）。这个部分最长可以为 16 B。下载程

序将信息连续地写入 ROM 存储器。因为 PIC18 芯片有 16 位的 ROM,所以这个字段的信息的存放顺序是先存放低位再存放高位。

```
LOC    OBJ             LINE
                       00003 LIST P=PIC18F8720,F=INHX32,N=0,ST=OFF,R=HEX
                             ;INTX32 for > 64KB
                       00004  #include P18F8720.INC
                       00001        LIST
             01306          LIST
22 02 0E 83            00005        CONFIG OSC=HS, OSCS=OFF
01 80 FF C0 FF E0 FF 40
                       00006        CONFIG BORV=45,PWRT=ON, BOR=ON
                       00007        CONFIG WDT=OFF
                       00008        CONFIG DEBUG=OFF, LVP=OFF, STVR=OFF

  00000007             00010 R1 equ 0x07
  00000008             00011 R2 equ 0x08
  00000009             00012 R3 equ 0x09

000000                 00014        ORG 0
000000 6A93            00015        CLRF   TRISB
000002 0E55            00016        MOVLW 0x55      ;WREG = 55h
000004 6E81            00017        MOVWF PORTB
000006 1E81            00018 L3     COMF   PORTB,F
000008 EC78  F094      00019        CALL   QDELAY
00000C D7FC            00020        BRA    L3

                       00023 ;-----------1/4 SECOND DELAY
0128F0                 00024        ORG 128F0H
0128F0                 00025 QDELAY
0128F0 0E02            00026        MOVLW D'2'
0128F2 6E07            00027        MOVWF R1
0128F4 0EFA            00028 D1     MOVLW D'250'
0128F6 6E08            00029        MOVWF R2
0128F8 0EFA            00030 D2     MOVLW D'250'
0128FA 6E09            00031        MOVWF R3
0128FC 0000            00032 D3     NOP
0128FE 0000            00033        NOP
012900 0609            00034        DECF   R3, F
012902 E1FC            00035        BNZ    D3
012904 0608            00036        DECF   R2, F
012906 E1F8            00037        BNZ    D2
012908 0607            00038        DECF   R1, F
01290A E1F4            00039        BNZ    D1
01290C 0012            00040        RETURN
                       00041        END
```

图 8-14　带有 INHX32 选项的测试程序的列表文件(注意 ORG 用于定义 QDELAY 的地址,为简化起见,一些注释和其他行已被删除)

(6) CC 是一个独立的字节。这最后的单字节是该行所有信息的校验和。校验和字节数用于错误检测。关于校验和字节数已在第 6 章和第 7 章中讨论过。注意,每行末尾的校验和字节数是该行所有信息的校验和,而不仅仅是数据部分的校验和。

例 8-5 分别分析图 8-13 中第 3 行和第 4 行的 6 个部分。

解:(a) 在第 3 行中,在起始符号":"之后,可知 BB=02,这意味着在这一行中有 2 B。AAAA=0000 及 TT=04 说明高 16 位地址由 HHHH 字段提供,所以由 HHHH 字段给出的 16 位地址为 000128F0H,即 0001。

(b) 在第 4 行中,在起始符号":"之后,可知 BB=10(十进制数为 16),这意味着在这一行中有 16 B。AAAA=28F0 是低 16 位的地址,也是信息写入的地方。TT=00 表明这不是最后一行。接下来是 16 B 的数据:020E076EFA0E086EFA0E096E00000000。最后一个字节数 56 是校验和。

8.3.3 Intel 十六进制分段文件(INHX8S)

INHX8S 选项又被称作 Intel 分段十六进制格式。当在 LIST 指令中启用 INHX8S 选项时,将会得到 2 个文件:用于低字节的 .hxl 文件和用于高字节的 .hxh 文件。因为 PIC18 ROM 是 16 位宽,所以将低字节存放到偶数地址,而将高字节存放到奇数地址,如图 8-15 所示。MPLAB 汇编器提供了 INHX8S 选项,是因为在许多带有外部存储器的 PIC18 芯片中,都需要将内存分为奇偶地址存储区以构成一个 16 位的 ROM 空间。注意到 ROM 芯片的引脚 D0~D7;因此,它们的存储结构是 Nk×8(如 64k×8)。这意味着必须将十六进制文件写入偶数存储区的 ROM 中,而将 .hxh 文件写入奇数地址的 ROM 中。.hxl 和 .hxh 格式同 INHX8M 一样,文件格式的每个存储区都不大于 64 KB。因此,使用分段 ROM 时,将得到一个最大为 128 KB 的 ROM,每个存储区为 64 KB。

320~322

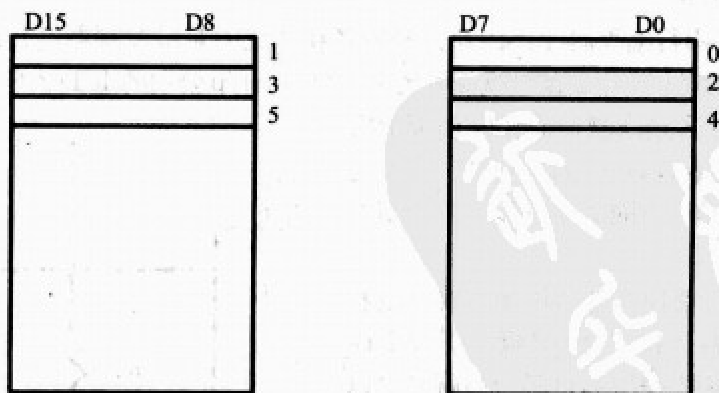


图 8-15 PIC18 外部存储器的奇数地址和偶数地址存储区

8.3.4 复习题

1. 判断对错:Intel 十六进制文件不使用校验和来保证数据完整性。
2. Intel 十六进制文件每行的第一个字节表示______。
3. Intel 十六进制文件每行的最后一个字节表示______。
4. Intel 十六进制文件的 TT 字段为 00,这意味着什么?
5. 计算下列数值的校验和:22H、76H、5FH、8CH、99H。
6. 将第 5 题中所有的数值和校验和加起来,将得到什么结果?

7. 判断对错：如果在 INHX32 文件中 TT 字段为 04，这说明记录是 32 位地址中的高 16 位。

8.4 PIC18 Trainer 的设计和装载

本节将讨论一个简单的基于 PIC18 的 Trainer 的连接。同时，还将介绍几种向 PIC 微控制器下载十六进制文件的方法。Microchip 设计的芯片在下载程序方面具有很大的灵活性。下面是 3 种下载装载程序的基本方法。

(1) 使用设备烧录器将程序下载到独立于系统的微控制器。这在生产平台上很有用，因为一组编程器可以在同一时间对许多芯片进行编程。大多数主流的设备烧录器都支持 PIC18 系列，Advin 和 EETools 是两个最著名的公司。Microchip 为所有产品都提供编程器，PICkit 2 和 PICSTART PLUS 就是其中的两个代表。请浏览 Microchip 公司的网站查看完整的编程器清单。由于设备编程器价格昂贵，用户也可以自行设计设备编程器，以减少购买编程器的开支。从零开始设计编程器已超出本书的范围，读者可以浏览专门的网站来获取相关的信息。

323

设备编程方法是比较简单直接的：即芯片在插入电路使用前需要编程，又或者，如果芯片在插座中，那么可以拔出来重新编程。ZIF(零插拔力)插座较标准插座拔插更快、损害更少。在拔除或者重插芯片时，必须观察 ESD(静电放电)过程。虽然 PIC 设备是抗震的，但是装卸它们仍然需要格外小心，以防损坏。这个方法允许在设计中使用所有的设备资源。同其他两种方法一样，本方法不共享引脚和芯片的内部资源。这样就允许嵌入式设计者要在设计中使用最少的电路板空间。

(2) 电路内部串行编程器(ICSP)允许开发者在系统中对微控制器进行编程和调试。在接受这种设置的系统上使用 2 条线缆就可实现。Microchip 公司的 ICD 2 是一个很好的程序调试设备。这种方法允许制造商在电路板上安装未编程的设备。这样，在产品送到顾客手中之前，厂家可以通过编程器向微控制器写入最新的文件资料。

电路内部串行编程非常适用于需要修改和周期性更新的设计。ICSP 使用两个引脚 RB7 和 RB6。当设备编程完毕后，这两个引脚又可以用作普通的 I/O 引脚。设计者必须确保这两个引脚同编程器没有冲突。对于 ICSP，MCLR 需要连接一个 10 kΩ 的上拉电阻。ICD 2 也需要连接电源 V_{dd} 和 Gnd 接地。设计者必须将引脚连接到电路板上的插座，以至于能同编程器相连接。图 8-16 给出了具体的引脚连接。设计者要在设计中仔细权衡这些方法的利弊。

324

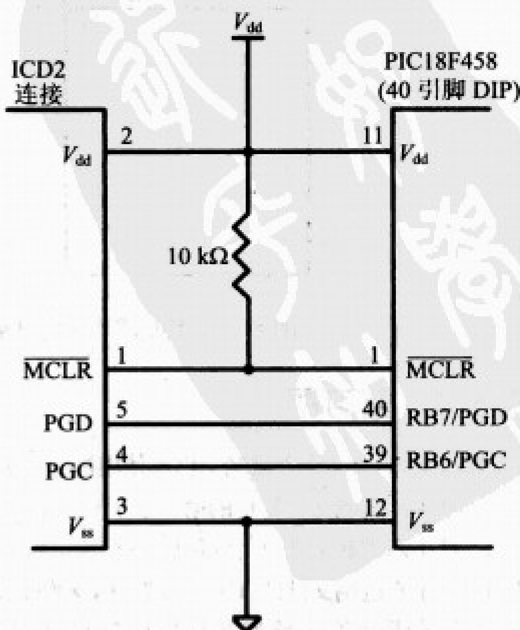


图 8-16 ICSP 连接

(3) 引导载入程序是烧录到微控制器程序 ROM 的一段代码，其目的是同用户的电路板通

信以下载程序。引导载入程序可以通过串行端口、CAN 端口、USB 端口甚至网络连接进行通信。引导载入程序还可以被设计来调试系统，这同 ICD 有些类似。这种编程方法对于没有设备编程器或可用的 ICD 的开发者特别有用。Microchip 公司在它的网站上给出了写引导载入程序的几种应用。使用引导载入程序的主要缺陷是它要求微控制器提供一个通信端口和程序代码存放的空间。同时，引导载入程序在使用前需要通过前面两种方法编程到设备中。

引导载入程序的方法适合于需要快速编程和测试代码的开发者。这种方法可以在没有 ICD 工具的情况下允许更新设备。开发者只需要一台带有和电路板端口兼容的电脑即可。(串行端口是很常用的一种端口，而 CAN 或是 USB 引导载入程序也可以被写入。)不过，这种方法会消耗大量的资源：如必须保留代码空间并加以保护，为了和 PC 通信需要使用外部设备等。使用这种方法开发项目可以帮助程序员方便地测试他们的代码。对于无需修改的成熟设计，使用其他两种方法则会更好些。

接下来将讨论使用诸如 PIC18F458/4580 和 PIC18F452/4520 芯片的基于 PIC18 系统的 ROM 载入程序有关的问题。同时，还将提供简单的 PIC18 Trainer 的设计指导。如果需要对它们进行绕线连接，请参阅附录 B 关于绕线的介绍。

8.4.1 基于 PIC18F452/458 的 Trainer

在基于 PIC18 微控制器的系统中，用户需要一个 ROM 烧录器将程序写入微控制器。对于 PIC18F 系列来说，除了将程序写入芯片之外，ROM 烧录器还可以擦除闪存原有的内容。如果是 PIC18C 系列，用户还需要一个 EPROM 擦除工具，因为它使用的是 UV-EPROM。在向 PIC18C 写入程序之前，用户需要使用 UV-EPROM 工具将它原有的内容擦除，这大概需要 20 min 的时间。而 PIC18F 不需要 UV-EPROM 工具，因为它具有闪存 ROM。

PIC18 闪存的容量

虽然所有的 PIC18 芯片有共同的特点，但是其芯片本身的 ROM 容量各有不同。表 8-14 给出了各种 PIC 芯片的片上 ROM 空间。想了解更多详情，请登录网站 http://www.microchip.com。注意，PIC18F2220 的片上 ROM 为 4 KB，PIC18F2410 的片上 ROM 是 16 KB，而 PIC18F458 的片上 ROM 为 32 KB。还要注意，PIC18F458 是 PIC18F452 代替品，只不过它多了一些额外的功能，比如 CAN。

表 8-14 PIC18 片上 ROM 的容量和地址空间

	片上程序 ROM (B)	程序地址范围 (十六进制)
PIC18F2220	4 K	00000～00FFF
PIC18F2410	16 K	00000～03FFF
PIC18F458/4580	32 K	00000～07FFF
PIC18F6680	64 K	00000～0FFFF
PIC18F8722	128 K	00000～1FFFF

325

例 8-6 计算下面每个芯片的 ROM 地址范围：

(a) 4 KB 的 PIC18F2220；

(b) 16 KB 的 PIC18F2410；

(c) 32 KB 的 PIC18F458/4580。

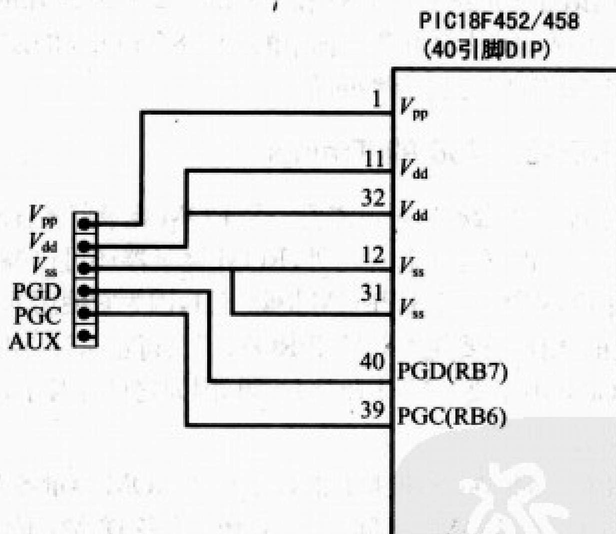
解:(a) 由于芯片 ROM 存储空间为 4 KB,总字节数为 4096 B(4×1024=4096),所以地址单元空间为 0000~0FFFH。注意,0 总是第一个地址单元。

(b) 由于芯片 ROM 存储空间为 32 KB,总字节数为 16 384 B(16×1024=16 384),所以地址单元空间为 0000~3FFFH。

(c) 32KB 的 ROM 有 32 768 B(32×1024=32 768)的存储空间。将 32 768 转换为十六进制,得到 8000H。因此,该芯片的 ROM 空间为 0000~7FFFH。

8.4.2 PIC18 Trainer 的连接

选择 PIC18F458 来设计 PIC18 Trainer,它可以让用户方便地绕接一个价格低廉但功能强大的训练装置,用于工作或家庭中。图 8-17 给出了基于 PIC18 的系统同 PICkit 2 编程器的连接。



注意:这种使用 PICkit2 插头的连接适用于 PIC 微控制器的所有系列。仅有的区别是引脚的数目和指定顺序。

图 8-17 PIC18F 通过 6 引脚的插头和 PICkit2 连接

326

PICkit2 是 Microchip 网站上一个价格较低的编程器。www.MicroDigitalEd.com 上给出了 PIC18 Trainer 连接的原理图。

8.4.3 PIC18 Trainer 程序下载

在设计了 PIC18 系统后,可以使用 PICkit2 的编程工具将程序下载到 Trainer 中,如图 8-18 所示。Microchip 不断地更新 MPLAB IDE 以支持 PICkit2 对所有 PIC 微控制器的编程功能。

8.4.4 汇编语言和 C 语言编写的 PIC18 测试程序

为了测试 PIC18 的硬件连接,可以运行一个简单的测试程序,将 PORTB 的所有位有间隔地在开关状态之间翻转,如程序 8-3 和程序 8-3C 所示。注意,在这些程序中,延迟时间是以

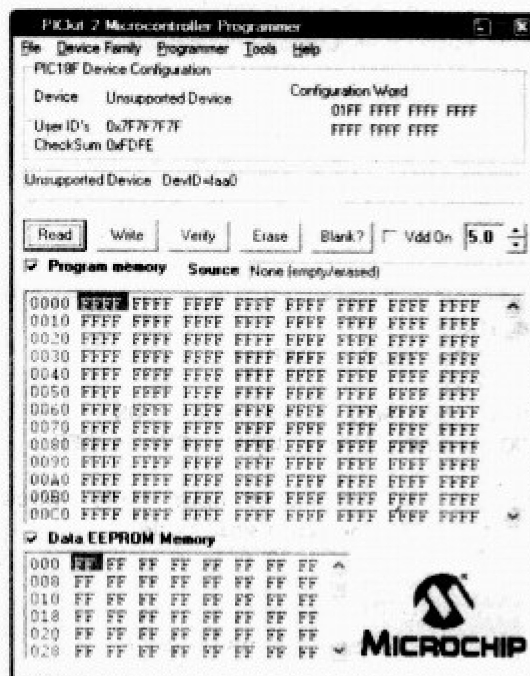


图 8-18 PICkit2 编程工具

10 MHz 的晶振为基础的。在开发自己的程序中,可以使用图 8-19 和图 8-20 的程序框架。

1. Trainer 的汇编语言测试程序

程序 8-3

```
;Program 8-3
      LIST P=PIC18F458, F=INHX32, N=0, ST=OFF, R=HEX
      #include P18F458.INC
      CONFIG OSC = HS, OSCS = OFF
      CONFIG WDT = OFF
      CONFIG BORV = 45, PWRT = ON, BOR = ON
      CONFIG DEBUG = OFF, LVP = OFF, STVR = OFF

R1    EQU   0x07
R2    EQU   0x08
R3    EQU   0x09

      ORG   0000H          ;note starting address
      CLRF  TRISB          ;make Port B an output port
      MOVLW 0x55           ;WREG = 55h
      MOVWF PORTB          ;put 55h on port B pins
L3    COMF  PORTB,F        ;toggle bits of Port B
      CALL  QDELAY         ;quarter of a second delay
      BRA   L3             ;continue

;-----------1/4 SECOND DELAY
QDELAY
      MOVLW D'2'
      MOVWF R1
D1    MOVLW D'250'
      MOVWF R2
D2    MOVLW D'250'
      MOVWF R3
```

```
D3    NOP
      NOP
      DECF   R3, F
      BNZ    D3
      DECF   R2, F
      BNZ    D2
      DECF   R1, F
      BNZ    D1
      RETURN
      END
```

```
#include P18F458.INC
     CONFIG OSC = HS, OSCS = OFF
     CONFIG WDT = OFF
     CONFIG BORV = 45, PWRT = ON, BOR = ON
     CONFIG DEBUG = OFF, LVP = OFF, STVR = OFF

     ORG    0000H          ;start of user code space
                           ;begin user code



                           ;end of user code
     END
```

328

图 8-19 用于MALAB的简单汇编语言框架

注意：这里没有使用LIST指令，因为它是MPLAB IDE的一个默认设置。

2. Trainer 的 C 语言测试程序

程序 8-3C

```
;Test Program 8-3C: Toggling PORTB for the PIC18F458/4580
;(452/4520) with XTAL = 10 MHz

      #pragma config OSC = HS, OSCS = OFF
      #pragma config PWRT = OFF, BOR = ON, BORV = 45
      #pragma config WDT = OFF, LVP = OFF
      #pragma config DEBUG = OFF, STVR = OFF

      #include <P18F458.h>

      void msdelay(unsigned int ms);
      void main(void)
        {
          TRISB = 0;                //make Port B an output
          while(1)
            {
            PORTB = 0x55;
            msdelay(500);
            PORTB = 0xAA;
            msdelay(500);
            }

        }

      //this is for a 10 MHz clock
      void msdelay(unsigned int ms)
```

```
{
unsigned int x;
unsigned char z;
for(x=0;x<ms;x++)
  for(z=0;z<165;z++);
}
```

```
#pragma config OSC = HS, OSCS = OFF
#pragma config PWRT = OFF, BOR = ON, BORV = 45
#pragma config WDT = OFF, LVP = OFF
#pragma config DEBUG = OFF, STVR = OFF

#include <P18F458.h>
void main(void)
  {




  }
```

图 8-20 用于 MPLAB 的简单 C 语言程序框架

329

8.4.5 故障检修的技巧

在 PIC18F458(或者 PIC18F452 系统)Trainer 上运行测试程序,应该能有延迟间隔地翻转所有的位。如果绕接系统不能工作,那么请按照下面的办法找出问题的所在。

(1) 切断电源,检查所有引脚的连接,特别是 V_{dd}和 GND。

(2) 使用示波器检查 MCLR(引脚 1)。当系统上电后,引脚 1 为高电平。在按动暂态开关时,它变为低电平。要确保暂态开关的正确连接。

(3) 在系统上电时,在示波器上观察 OSC1 引脚。如果正常,应该看到原始的正弦波形。这表明晶体振荡器在正常运行。

(4) 如果以上几个步骤都没有检测出故障,那么检查芯片上 ROM 的内容。它必须和 .lst 文件提供的操作码一致。汇编器提供的 .lst 文件将在汇编指令的左边列出操作码和操作数。如果在向片上 ROM 烧录和下载程序时没有出错,那么 .lst 文件所列的操作码和操作数必须同片上 ROM 的内容相一致。

8.4.6 复习题

1. 哪一种编程 PIC 微控制器的方法最适合大规模电路板生产?
2. 哪种方法允许进行系统调试?
3. 哪种方法可以允许小公司开发样机(原型),并为各种用户测试嵌入式系统?
4. 判断对错:PIC18C 带有 Flash 程序 ROM。
5. 在 PIC18F458/4580 中,哪个引脚用来复位?
6. 如果复位引脚没有被激活,那么它将处于什么状态?
7. PIC18F458/4580 使用的是哪一种 ROM?

8. 判断对错:PIC18只能将Intel十六进制格式的文件下载到ROM中。

9. 请列举两个理由,说明PIC18F芯片比PIC18C芯片更好用。

330

关于PIC18 Trainer,请登录下面的网站:http://www.MicroDigitalEd.com。

小结

这一章首先描述了PIC18F458芯片各个引脚的功能,并重点讨论了PIC18F458的CONFIG寄存器。CONFIG寄存器的地址从300001H开始,它们的地址不在程序ROM的地址范围内。它们将和应用程序一起被写入PIC芯片中。使用CONFIG寄存器可以开启一些功能,如低功率频率和看门狗定时器。本章还解释了Intel十六进制文件格式INHX8M和INHX32,讨论了INHX32格式如何使用32位的ROM地址、INHX8M如何使用16位地址。最后,还给出了PIC18 Trainer的设计。

习题

1. PIC18F458 DIP封装是一种_____引脚的封装。
2. V_{cc}和GND对应着哪两个引脚?
3. 在PIC18F458中,有多少个引脚是I/O端口引脚?
4. 晶体振荡器连接引脚_____和_____。
5. 如果PIC18F458的频率是40 MHz,那么它能连接的最大频率是多少?
6. 指出在DIP封装中分配给MCLR的引脚号。
7. MCLR是指_____。
8. MCLR引脚通常是处于_____(低电平,高电平),它需要一个_____(低电平,高电平)信号来激活。
9. PIC18F458复位后,PC(程序计数器)的内容是什么?
10. PIC18F458复位后,SP寄存器的内容是什么?
11. PIC18F458复位后,WREG寄存器的内容是什么?
12. PIC18F458复位后,TRIS寄存器的内容是什么?
13. 在PIC18F458中,有多少个引脚用作V_{dd}?
14. 在PIC18F458中,有多少个引脚用作V_{ss}(GND)?
15. 在OSC引脚中,哪一个引脚同PORTA位共享呢?
16. OSC 1和OSC 2是_____(输入,输出)引脚。
17. MCLR是一个_____(输入,输出)引脚。
18. 在DIP封装中,有多少引脚分配给PORTA,分别是哪些引脚?
19. 在DIP封装中,有多少引脚分配给PORTB,分别是哪些引脚?
20. 在DIP封装中,有多少引脚分配给PORTC,分别是哪些引脚?
21. 在DIP封装中,有多少引脚分配给PORTD,分别是哪些引脚? 331
22. 在系统复位后,端口的所有位被设置成_____(输入,输出)。
23. 在PIC18F458中,哪个端口只有3个引脚?
24. PIC18F458的哪个引脚没有复用功能而只能作为I/O引脚使用?
25. 判断对错:在PIC18F的复位状态,CPU没有执行任何代码。

26. 判断对错:在系统上电后,上电定时器(PWRT)和振荡器起动定时器(OST)让 PIC18 处在复位状态,直到系统电压和频率进入稳定状态。
27. 判断对错:上电定时器(PWRT)和振荡器起动定时器(OST)是 PIC18 必需的外部元件。
28. 判断对错:看门狗定时器是一个必需的外部元件。
29. 判断对错:如果在源代码中没有给出 CONFIG 值,PIC18 使用默认的 CONFIG 值。
30. 判断对错:CONFIG 寄存器使用和程序 ROM 相同的地址空间。
31. 请写出 CONFIG1H、CONFIG2L、CONFIG2H 和 CONFIG4L 的 ROM 地址。
32. CONFIG 寄存器是______位的。
33. 哪个 CONFIG 寄存器用于设定 PIC18F458 的时钟频率?
34. 哪个 CONFIG 寄存器用于设定 PIC18F458 的掉电复位电压?
35. 哪个 CONFIG 寄存器可以禁止看门狗定时器?
36. 如果掉电复位电压设为 4.2 V,对系统来说意味着什么?
37. 请写出对于 PIC18F458 系统带有下列选项的 CONFIG 指令:
    (a) OSC1—OSC2 连接到 20 MHz 的时钟,它是系统的唯一时钟源。
    (b) 将掉电电压设为 4.2 V,且启用上电定时器。
    (c) 没有看门狗定时器。
    (d) 没有栈溢出,没有背景调试程序,没有 LVP。
38. 对于 CONFIG1H,OSC 频率为哪一选项对应的系统消耗最小?
39. 哪个 CONFIG 寄存器用于设定 PIC18F458 的时钟源?
40. 计算下列在连接到 OSC1 和 OSC2 的晶体频率下的指令周期。假设都选择 HS 模式。
    (a) 12 MHz (b) 20 MHz (c) 25 MHz (d) 30 MHz
41. 判断对错:INHX32 选项可以通过 MPLAB 设置,而无需使用 LIST 指令。
42. 判断对错:INHX32 选项可以用于 ROM 空间大于 64 KB 的芯片。
43. 判断对错:INHX8M 选项可以用于 ROM 空间大于 64 KB 的芯片。
44. 判断对错:INHX8M 选项可以用于 ROM 空间小于 64 KB 的芯片。
45. 判断对错:INHX32 选项可以用于 ROM 空间为任意大小的芯片。
46. 分析图 8-10 中第一行的 6 个部分。
47. 验证图 8-10 中第一行的校验和,并判断信息是否出错。
48. 验证图 8-13 中第二行的校验和,并判断信息是否出错。
49. INHX8M 和 INHX32 十六进制文件的区别在哪里?
50. 分析图 8-13 中的 INHX32 Intel 十六进制文件。
51. 判断对错:使用 PICkit2 时,必须将 PIC18F 芯片从系统中拔出,并将它放到编程器中。
52. 判断对错:PICkit2 只能用于闪存芯片。
53. 下面哪种选择价格会便宜些?
    (a) MPLAB ICD2 (b) PICkit2
54. 编制程序:从 PORTB 读入 8 位数据并将它送到端口 PORTC 和 PORTD。
55. 编制程序:从 PORTD 读入 8 位数据并将它送到端口 PORTB 和 PORTC。
56. 端口 PORTB 的哪个引脚是 PGD(程序数据),哪个引脚是 PGC(程序时钟)?
57. 在系统复位时,PIC18F458 在哪个程序地址被唤醒? 其含义是什么?
58. 编制程序,连续地翻转 PORTB 的各位。
    (a) 使用 AAH 和 55H (b) 使用 COMF 指令

332

59. 对于PIC18F458,程序ROM的最后一个单元的地址是什么?
60. 对于PIC18F8722,程序ROM的最后一个单元的地址是什么?
61. 对于PIC18F452,程序ROM的最后一个单元的地址是什么?

复习题答案

8.1 节

1. 1　2. 000000　3. 000000　4. 低电平　5. 两个V_{dd}引脚和两个GND引脚。

8.2 节

333
1. 16 MHz/4=4 MHz,1/4 MHz=250 ns。
2. 300001H(十六进制)　3. 错误。　4. 0,200　5. CONFIG2H,300003H　6. 正确。
7. 错误。　8. 正确。　9. 错误。　10. 4.2 V。

8.3 节

1. 错误。　2. 该行数据的字节数。　3. 该行所有字节的校验和。
4. 00指这不是最后一行,后面还有更多的行。
5. 22H+76H+5FH+8CH+99H=21CH。舍去进位,可以得到1CH,它的2进制补码为E4H。
6. 22H+76H+5FH+8CH+99H+E4H=300H。舍去进位,可以得到00,这说明数据没有出错。
7. 正确。

8.4 节

1. 设备烧录器　2. 电路内部串行调试器　3. ICSP　4. 错误。　5. 引脚1。　6. 高电平。
334
7. Flash　8. 正确。　9. 可以同ICSP一起使用,具有更快的开发时间。

第9章 PIC18定时器的汇编编程和C编程

学习目标：

- ☐PIC18的定时器及其寄存器
- ☐PIC18定时器的不同模式
- ☐PIC18定时器用于时延的汇编编程和C编程
- ☐PIC18计数器用作事件计数器的汇编编程和C编程

PIC18依具体型号有2～5个不等的定时器，即定时器0、定时器1、定时器2、定时器3和定时器4。它们可以用作定时器来产生时延，或者用作计数器来计数微控制器的外部事件。9.1节将介绍如何使用定时器0和定时器1产生时延。9.2节将介绍如何使用它们来作为事件计数器。9.3节将讨论如何使用C语言来编程PIC18定时器。定时器2和定时器3将会在9.4节中讨论。

9.1 定时器0和定时器1编程

每个定时器都需要靠一个时钟脉冲来推动。时钟源可以是内部的也可以是外部的。如果使用内部时钟源，那么连接到OSC1和OSC2引脚的晶振频率的1/4频率信号($F_{osc}/4$)将被输入给定时器。它可以用来产生时延，因此又被称作定时器。若选用外部时钟源，则可以通过PIC18的一个引脚输入脉冲，这就叫作计数器。在本节中，将讨论PIC18定时器，而在下一节将讨论定时器作为计数器的编程。

9.1.1 定时器的基本寄存器

大部分PIC18定时器都是16位的。因为PIC18是8位体系结构，所以每个16位定时器将被当作两个独立的寄存器来访问，即低字节寄存器(TMRxL)和高字节寄存器(TMRxH)。每个定时器都有TCON(定时器控制)寄存器来设置工作模式。下面分别讨论每个定时器。

9.1.2 定时器0寄存器和编程

定时器0可以用作8位或者16位的定时器。定时器0的16位寄存器将使用高字节和低字节来访问，如图9-1所示。低字节寄存器被称作TMR0L(定时器0的低字节)，而高字节寄存器被称作TMR0H(定时器0的高字节)。这些寄存器的访问方法和其他SFR相同。例如，指令`MOVWF TMR0L`把WREG寄存器的内容传送入TMR0L，即定时器0的低字节。读取这

些寄存器的方法也和其他寄存器一样。例如，指令 MOVFF TMR0L，PORTB 把 TMR0L（寄存器 0 低字节）的内容传送到 PORTB 中。

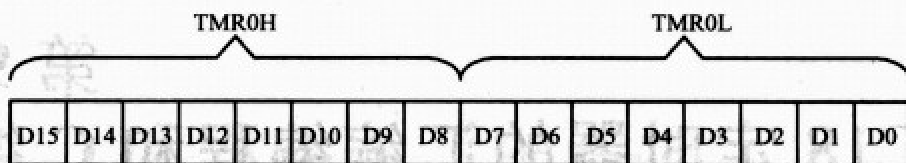


图 9-1 定时器 0 的高字节和低字节寄存器

9.1.3 T0CON（定时器 0 控制）寄存器

336

每个定时器都有一个被称作 TCON 的控制寄存器，用来设置定时器的各种工作模式。T0CON 是 8 位的寄存器，用来控制定时器 0。T0CON 的位表示如图 9-2 所示。

TMR0ON	T08BIT	T0CS	T0SE	PSA	T0PS2	T0PS1	T0PS0

TMR0ON	D7	定时器 0 的开关控制位
		1＝使能（启用）定时器 0
		0＝停止定时器
T08BIT	D6	定时器 0 的 8 位/16 位选择位
		1＝定时器 0 用作 8 位定时器/计数器
		0＝定时器 0 用作 16 位定时器/计数器
T0CS	D5	定时器 0 的时钟源选择位
		1＝外部时钟，来自 RA4/T0CK1 引脚
		0＝内部时钟，来自 XTAL 晶振的 $F_{osc}/4$
T0SE	D4	定时器 0 的信号源边沿选择位
		1＝在 T0CK1 引脚的下降（由高到低）沿加 1
		0＝在 T0CK1 引脚的上升（由低到高）沿加 1
PSA	D3	定时器 0 的预分频器选择位
		1＝定时器 0 时钟输入不经过预分频器
		0＝定时器 0 时钟输入来自预分频器输出
T0PS2：T0PS0	D2D1D0	定时器 0 的预分频器值选择位
	000＝1：2	预分频值（$F_{osc}/4/2$）
	001＝1：4	预分频值（$F_{osc}/4/4$）
	010＝1：8	预分频值（$F_{osc}/4/8$）
	011＝1：16	预分频值（$F_{osc}/4/16$）
	100＝1：32	预分频值（$F_{osc}/4/32$）
	101＝1：64	预分频值（$F_{osc}/4/64$）
	110＝1：128	预分频值（$F_{osc}/4/128$）
	111＝1：256	预分频值（$F_{osc}/4/256$）

图 9-2 T0CON（定时器 0 控制）寄存器

T0CS（定时器0时钟源）

在T0CON寄存器中，该位用来决定到底使用内部时钟源（$F_{osc}/4$）还是外部时钟源。若T0CS=0，则使用$F_{osc}/4$作为时钟源。在这种情况下，定时器通常用来产生时延，如例9-1所示。若T0CS=1，则时钟源是外部的，来自RA4/T0CKI，也就是PIC1818F4580/4520的DIP封装上的引脚6。当时钟源来自外部时，定时器可用作事件计数器。定时器的计数功能将在下节讨论。请看例9-2。

例9-1 请确定满足下面条件的T0CON的值：将定时器0编程为16位模式，不使用预分频器。使用PIC18的$F_{osc}/4$晶振作为时钟源，在上升沿时计数加1。

解：

T0CON=0000 1000 16位，$F_{osc}/4$时钟源，不使用预分频器，定时器0关闭。

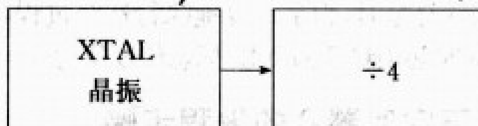
337

例9-2 在下面不同晶振的PIC18系统中，确定定时器的时钟频率和周期。假设没有使用预分频器。

(a) 10 MHz (b) 16 MHz (c) 4 MHz

解：

(a) 1/4 × 10MHz = 2.5 MHz， T = 1/2.5MHz = 0.4μs
(b) 1/4 × 16MHz = 4MHz， T = 1/4MHz = 0.25μs
(c) 1/4 × 4MHz = 1MHz， T = 1/1MHz = 1μs



注意：PIC18定时器在预分频器的基础上使用晶振频率的1/4。

9.1.4 TMR0IF标志位

注意，TMR0IF位（定时器0的中断标志）是INTCON（中断控制）寄存器的一部分，如图9-3所示。关于INCON寄存器的其他功能将会在第11章讨论。当定时器达到最大值FFFFH时，将会返回到0000，并且TMR0IF将被置1，如图9-4所示。第11章将会介绍如何使用TMR0IF来产生中断。下面将介绍定时器0的16位工作模式。

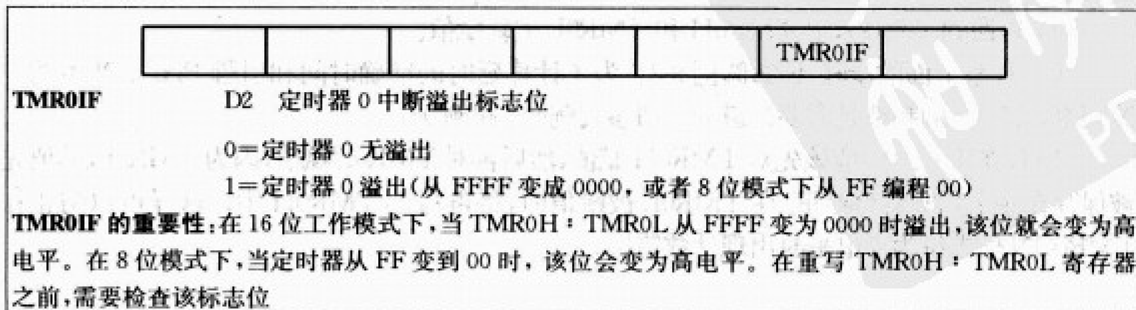


图9-3 INTCON（中断控制寄存器）的TMR0IF标志位

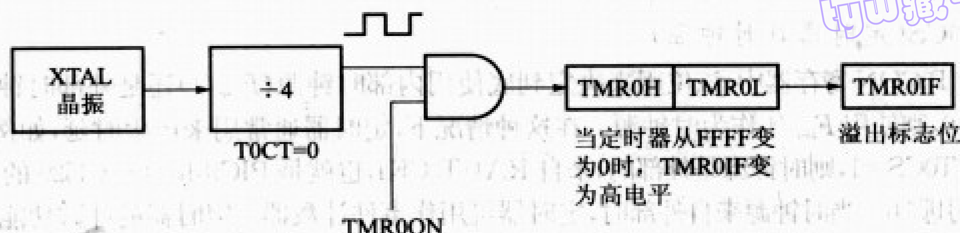


图9-4　定时器0溢出标志位

338

9.1.5　16位定时器编程

下面是16位工作模式的特性和操作。

(1) 这是16位的定时器，所以，允许将0000～FFFFH之间的值下载到寄存器TMR0H和TMR0L。

(2) 在TMR0H和TMR0L装载了16位的初值后，必须启动定时器。对于定时器0，可以使用指令BSF T0CON, TMR0ON来完成。

(3) 在定时器被启动后，它开始数数。定时器一直计数到上限FFFFH。当它从FFFFH又变为0000时，标志位TMR0IF(定时器中断标志位，是INTCON寄存器的一部分)变为高电平。该定时器标志位是可监视的。当定时器标志位变为高电平时，可以选择停止定时器。

(4) 当定时器到达上限并复零时，为了重复地执行这个过程，必须对寄存器TMR0H和TMR0L重新加载初始值，同时必须将TMR0IF标志位复位为0。

9.1.6　在16位模式下定时器0的编程步骤

为了使用定时器0的16位模式来产生一个时延，可以采取以下步骤。

(1) 对T0CON寄存器赋值，说明使用的工作模式(8位还是16位)和预分频器选项。

(2) 依次对TMR0H和TMR0L赋计数初值。

(3) 使用指令BSF T0CON, TMR0ON启动定时器。

(4) 持续监视定时器标志位(TMR0IF)，观察它是否变为高电平。如果TMR0IF变为高电平，跳出循环。

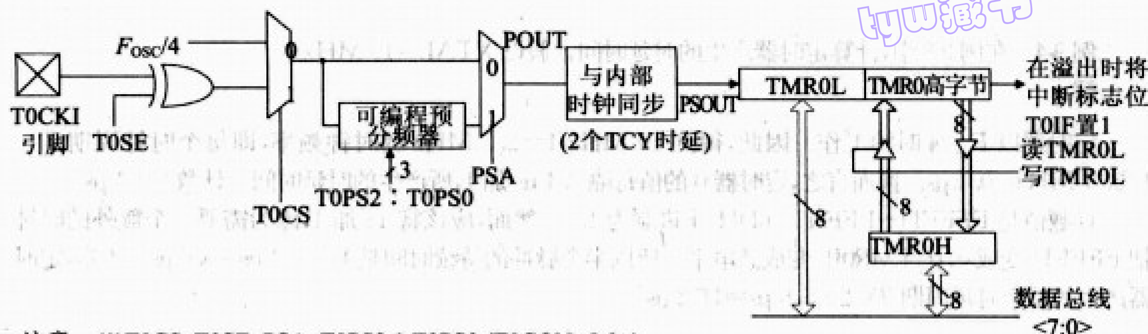
(5) 使用指令BCF T0CON, TMR0ON停止定时器的工作。

(6) 为开始新一轮的循环，清除TMR0IF标志位。

(7) 返回到步骤(2)，对TMR0H和TMR0L重新赋值。

为了理解上面的步骤，请参阅例9-3。为了计算延时的准确时间和引脚PB5上产生的方波频率，XTAL的频率是需要知道的。请参阅例9-4和例9-5。

注意，在图9-5中，应该先对TMR0H赋值，然后再对TMR0L赋值，因为TMR0H的值先被保存在一个临时寄存器里，在TMR0L被赋值时，将再写入TMR0H中。这样可以防止在TMR0ON标志位为高电平时出现计数错误。



注意：(1)T0CS, T0SE, PSA, T0PS2：T0PS0 (T0CON<5:0>)。
(2)在复位期间，定时器0被设为8位模式，时钟输入来自T0CKI，预分频器为最大值。

图 9-5 定时器 0 的 16 位方框图

339

例 9-3 在下面的程序中，欲在 PORTB.5 位产生一个占空比为 50%的方波(高低电平各占半个周期)。使用定时器 0 来生成时延。请分析这个程序。

```
        BCF    TRISB,5          ;PB5 as an output
        MOVLW  0x08             ;Timer0,16-bit,int clk,no prescale
        MOVWF  T0CON            ;load T0CON reg.
HERE    MOVLW  0xFF             ;TMR0H = FFH, the high byte
        MOVWF  TMR0H            ;load Timer0 high byte
        MOVLW  0xF2             ;TMR0L = F2H, the low byte
        MOVWF  TMR0L            ;load Timer0 low byte
        BCF    INTCON, TMR0IF   ;clear timer interrupt flag bit
        BTG    PORTB,5          ;toggle PB5
        BSF    T0CON, TMR0ON    ;start Timer0
AGAIN   BTFSS  INTCON, TMR0IF   ;monitor Timer0 flag until
        BRA    AGAIN            ;it rolls over
        BCF    T0CON, TMR0ON    ;stop Timer0
        BRA    HERE             ;load TH, TL again
```

解：

上述程序的设计步骤如下。

(1) 对 T0CON 寄存器赋值。

(2) 将 FFF2H 赋值给 TMR0H 和 TMR0L。

(3) 使用指令 `BCF INTCON, TMR0IF` 对定时器的中断标志位清零。

(4) 翻转 PORTB.5，得到脉冲的高低电平。

(5) 使用指令 `BSF T0CON, TMR0ON` 启动定时器 0。

(6) 定时器 0 对来自晶振的每个时钟信号计数。在定时器计数时，它历经的状态为 FFF3、FFF4、FFF5、FFF6、FFF7、FFF8、FFF9、FFFA、FFFB，以此类推，直到 FFFF。再经过一个时钟，定时器将变为 0，定时器标志位变为高电平(TMR0IF=1)。此时，执行指令 `BTFSS INTCON, TMR0IF` 就跳过了指令 `BRA AGAIN`。

(7) 使用指令 `BCF T0CON, TMR0ON` 终止定时器 0，再重复上述的过程。

注意，为了重复该过程，必须重新对 TMR0L 和 TMR0H 寄存器赋值，而且要重新启动定时器。



340

例 9-4 在例 9-3 中,计算定时器产生的时延时间。假设 XTAL=10 MHz。

解:

定时器以 $F_{osc}/4$ 时钟工作。因此,得到 10 MHz/4=2.5 MHz 的时钟频率,即每个时钟周期 $T=1/2.5$ MHz=0.4 μs。换而言之,定时器 0 的值每隔 0.4 μs 加 1,所产生的时延时间=计数×0.4 μs。

计数值是 FFFFH−FFF2H=0DH(十进制为 13)。然而,应该将 13 加 1,因为需要一个额外的时钟把 FFFFH 变成 0,让 TMR0IF 变成高电平。所以半个脉冲的持续时间是 14×0.4 μs=5.6 μs。于是,定时器产生的整个时延周期 $T=2\times5.6$ μs=11.2 μs。

例 9-5 计算在 PORTB.5 引脚上产生的方波频率。

解:

为了得到更精确的时间,需要加上循环体中的一些指令的时钟周期。

```
                                                     周期数
             BCF    TRISB,5
             MOVLW  0x08
             MOVWF  T0CON
             BCF    INTCON, TMR0IF

HERE         MOVLW  0xFF                               1
             MOVWF  TMR0H                              1
             MOVLW  -D'48'                             1
             MOVWF  TMR0L                              1
             CALL   DELAY                              1
             BTG    PORTB,5                            1
             BRA    HERE                               1
;——————————delay using Timer0
DELAY        BSF    T0CON, TMR0ON                      1
AGAIN        BTFSS  INTCON, TMR0IF                     1
             BRA    AGAIN                              1
             BCF    T0CON, TMR0ON                      1
             BCF    INTCON, TMR0IF                     1
             RETURN                                    1
                                                      13
```

341

因此,$T=2\times(48+13)\times0.4$ μs=48.8 μs,$F=20.491$ kHz。

对于晶振频率 XTAL=10 MHz,使用定时器的 16 位模式时,可以设计出用于时延计算的公式,如图 9-6 所示。微软 Windows 附件中的科学计算器也可以用来计算 TMR0H 和 TMR0L 的值。该计算器支持十进制、十六进制和二进制的计算。请参阅例 9-6 和例 9-7。

(FFFF - YYXX + 1) × 0.4 μs where YYXX are the TMR0H, TMR0L initial values respectively. Notice that YYXX values are in hex.	Convert YYXX values of the TMR0H, TMR0L register to decimal to get a NNNNN decimal number, then (65536 - NNNNN) × 0.4 μs
(a) 十六进制	(b) 十进制

图 9-6 XTAL=10 MHz 的时延计算,不使用预分频器

例9-6 使用图9-6中的两种方法，计算在下面的代码中定时器0产生的时延。不包括循环的指令消耗。

```
        BCF    TRISB,5          ;PB5 as an output
        MOVLW  0x80             ;Timer0,16-bit,int clk, no prescale
        MOVWF  T0CON
        BCF    INTCON,TMR0IF    ;clear Timer0 interrupt
HERE    MOVLW  0xB8             ;TMR0H = B8, the high byte
        MOVWF  TMR0H
        MOVLW  0x3E             ;TMR0L = 3E, the low byte
        MOVWF  TMR0L
        BSF    T0CON, TMR0ON    ;start Timer0
AGAIN   BTFSS  INTCON, TMR0IF   ;monitor Timer0 flag until
        BRA    AGAIN            ;it rolls over
        BCF    T0CON, TMR0ON    ;stop Timer0
        BCF    INTCON, TMR0IF   ;clear Timer0 interrupt
        BTG    PORTB,5          ;toggle PB5
        BRA    HERE             ;load TH, TL again
```

解：

(a) (FFFF−B83E+1)=47C2H=18 370(十进制)，18 370×0.4 μs=7.348 ms。

(b) 因为TMR0H：TMR0L=B83EH=47166(十进制)，得到65 536−47 166=18 370。即定时器从B83EH计数到FFFFH。计数和归零过程共历经了18 370个时钟周期，每个时钟周期的持续时间为0.4 μs。因此，可以得到脉冲宽度为18 370×0.4 μs=7.348 ms。

342

例9-7 计算下面的程序产生的方波频率，假设XTAL=10 MHz。在计算中不包括循环的指令消耗。

```
        BCF    TRISB,5          ;PB5 as an output
        MOVLW  0x08             ;Timer0,16-bit,int clk,no prescale
        MOVWF  T0CON            ;load T0CON reg.
HERE    MOVLW  0x76             ;TMR0H = 76H, the high byte
        MOVWF  TMR0H            ;load Timer0 high byte
        MOVLW  0x34             ;TMR0L = 34H, the low byte
        MOVWF  TMR0L            ;load Timer0 low byte
        BCF    INTCON,TMR0IF    ;clear timer interrupt flag bit
        CALL   DELAY
        BTG    PORTB,5          ;toggle PB5
        BRA    HERE             ;load TH, TL again
;————————delay using Timer0
DELAY   BSF    T0CON,TMR0ON     ;start Timer0
AGAIN   BTFSS  INTCON,TMR0IF    ;monitor Timer0 flag until
        BRA    AGAIN            ;it rolls over
        BCF    T0CON,TMR0ON     ;stop Timer0
        RETURN
```

解：

因为FFFFH−7634H=89CBH+1=89CCH，而89CCH=35 276个时钟周期，所以有35 276×0.4 μs=14.11 ms，频率=1/(14.11 ms×2)=35.434 Hz。在这个计算中，不包括循环的指令消耗。

9.1.7 计算定时器的载入值

假设已知期望的时延，如何计算寄存器TMR0H和TMR0L的载入值呢？要计算寄存器

TMR0H和TMR0L的载入值,请看例9-8和例9-9,这里使用晶振为10 MHz的PIC18系统。

假设XTAL=10 MHz,不使用预分频器。可以根据以下步骤计算寄存器TMR0H和TMR0L的值。

(1) 将期望的时延值除以0.4 μs。

(2) 执行65 536−n,其中n是在步骤(1)中得到的十进制数。

(3) 把在步骤(2)中得到的结果转换为十六进制数,$yyxx$就是即将载入定时器的寄存器的初始十六进制值。

343

(4) 设置TMR0L=xx,TMR0H=yy。

例9-8 假设XTAL=10 MHz,编制程序,在引脚PORTB.3产生一个周期为10 ms的方波。

解:

对于T=10 ms的方波,时延必须为5 ms。因为XTAL=10 MHz,定时器每隔0.4 μs加1。也就是说,需要5 ms/0.4 μs=12 500个时钟。于是,有65 536−12 5000=53 036=CF2CH。因此,可以得到TMR0H=CF,TMR0L=2C。

```
        BCF    TRISB,3          ;PB3 as an output
        MOVLW  0x08             ;Timer0,16-bit,int clk,no prescale
        MOVWF  T0CON            ;load T0CON reg
HERE    MOVLW  0xCF             ;TMR0H = CFH, the high byte
        MOVWF  TMR0H            ;load Timer0 high byte
        MOVLW  0x2C             ;TMR0L = 2CH, the low byte
        MOVWF  TMR0L            ;load Timer0 low byte
        BCF    INTCON,TMR0IF    ;clear timer interrupt flag bit
        CALL   DELAY
        BTG    PORTB,3          ;toggle PB3
        BRA    HERE             ;load TH, TL again
;----------delay using Timer0
DELAY BSF    T0CON,TMR0ON     ;start Timer0
AGAIN BTFSS  INTCON,TMR0IF    ;monitor Timer0 flag until
        BRA    AGAIN            ;it rolls over
        BCF    T0CON,TMR0ON     ;stop Timer0
        RETURN
```

例9-9 假设XTAL=10 MHz,修改例9-8中的程序,在引脚PORTB.3产生一个频率为2 kHz的方波。

解:

请看下面的步骤。

(a) $T=1/F=1/2$ kHz=500 μs,这就是方波的周期。

(b) 高低电平的持续时间都是方波周期的1/2,等于250 μs。

(c) 有250 μs/0.4 μs=625,65 536−625=64 911,对应的十六进制值是FD8FH。

344

(d) 因此,TMR0L=8FH,TMR0H=FDH,都是十六进制形式。

例9-10 修改例9-8中TMR0L和TMR0H的值,以产生最大的时延。请计算该时延(单位为ms)。在计算中,忽略循环的指令消耗。

解：

为了得到最大的时延，将TL和TH都置0。定时器将从0000计数到FFFFH，再归为零。

```
        BCF   TRISB,3             ;PB3 as an output
        MOVLW 0x80                ;Timer0,16-bit,int clk,no prescale
        MOVWF T0CON               ;load T0CON reg.
HERE    CLRF  TMR0H
        CLRF  TMR0L               ;TH = TL = 0
        BCF   INTCON,TMR0IF       ;clear timer interrupt flag bit
        CALL  DELAY
        BTG   PORTB,3             ;toggle PB3
        BRA   HERE                ;load TH, TL again
;————————delay using Timer0
DELAY   BSF   T0CON,TMR0ON        ;start Timer0
AGAIN   BTFSS INTCON,TMR0IF       ;monitor Timer0 flag until
        BRA   AGAIN               ;it rolls over
        BCF   T0CON,TMR0ON        ;stop Timer0
        RETURN
```

将TL和TH都置0，即让定时器从0000计数到FFFFH，再归零，从而使TMR0IF标志位变为高电平。这就历经了65 536个状态，因此得到的时延=(65 536−0)×0.4 μs=26 214 ms。于是，得到的最小频率1/(2×26.214 ms)=1/(52.428 ms)=19.073 Hz。

9.1.8 使用Windows计算器寻找TH和TL

微软Windows中的科学计算器是一个唾手可得而又简单易用的工具，可以用来寻找TMR0H值和TMR0L值。假设需要计算35 000个周期为0.4 μs的时钟所产生的时延的TMR0H值和TMR0L值。计算过程如下。

(1) 打开微软Windows中的科学计算器，并选择十进制方式。

(2) 输入35 000。

(3) 选择十六进制方式，将35 000转换为十六进制的88B8H。

(4) 选择"+/−"得到十进制的−35 000(十六进制为7748H)。

(5) 该十六进制数的最低两位(48)用于TMR0L，而另外的两位(77)用于TMR0H。因为使用的是十六位数，所以忽略左边所有的F。

345

9.1.9 预分频器和长时延的产生

正如在已介绍的例子中看到的，时延的长短由两个因素决定：晶振频率和定时器的16位寄存器。这两个因素都是不受PIC18程序员控制的。在例9-10中已看到，为了得到最长的时延，将TMR0H和TMR0L都置为0。如果这个时延时间还不够呢？可以使用T0CON寄存器中的预分频器选项来减少周期，增加时延。如图9-2所示，使用T0CON的预分频器选项可以将指令时钟分成2～256份不等。

到目前为止，在没有使用预分频器时，晶振频率都是除以4($F_{osc}/4$)然后送给定时器0。如果启用T0CON寄存器的预分频器位，那么会将指令周期($F_{osc}/4$)再分后才送入定时器0。T0CON寄存器的低3位用于选择预分频值。如图9-2所示，预分频器值可以是2、4、8、16、32、64等。注意，最小值是2，最大值是256。例9-11～例9-15说明了如何编程预分频器选项。

例 9-11 如果定时器0工作在16位模式，使用预分频器值64，时钟源为内部时钟($F_{osc}/4$)，上升沿计数。请确定T0CON的值。

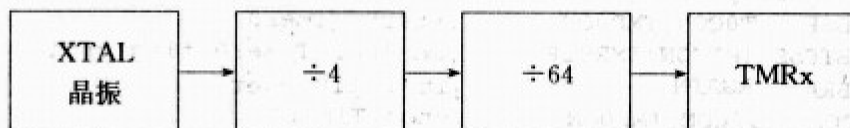
解：

从图9-2可知，T0CON=0000 0101；定时器0工作在16位模式，XTAL时钟源，预分频器值是64。

例 9-12 计算使用下面不同频率的PIC18系统的定时器的时钟频率和周期。假设预分频器值是1∶64。

(a) 10 MHz　　　　　　(b) 16 MHz

解：



(a) 由于预分频器值为1∶64，所以有1/64×2.5 MHz=39 062.5 Hz，1/4×10 MHz=2.5 MHz，T=1/39 062.5 Hz=25.6 μs。

346

(b) 由于预分频器值为1∶64，所以有1/64×4 MHz=62 500 Hz，1/4×16 MHz=4 MHz，T=1/62 500 Hz=16 μs。

例 9-13 研究下面的程序，计算其时延时间(单位为s)。忽略用于循环的指令消耗。假设XTAL=10 MHz。

```
      BCF    TRISB,2          ;PB2 as an output
      MOVLW  0x05             ;Timer0,16-bit,int clk,prescaler 64
      MOVWF  T0CON            ;load T0CON reg.
HERE  MOVLW  0x01             ;TMR0H = 01H, the high byte
      MOVWF  TMR0H            ;load Timer0 high byte
      MOVLW  0x08             ;TMR0L = 08H, the low byte
      MOVWF  TMR0L            ;load Timer0 low byte
      BCF    INTCON,TMR0IF    ;clear timer interrupt flag bit
      CALL   DELAY
      BTG    PORTB,2          ;toggle PB2
      BRA    HERE             ;load TH, TL again
;----------delay using Timer0
DELAY BSF    T0CON,TMR0ON     ;start Timer0
AGAIN BTFSS  INTCON,TMR0IF    ;monitor Timer0 flag until
      BRA    AGAIN            ;it rolls over
      BCF    T0CON,TMR0ON     ;stop Timer0
      RETURN
```

解：

TMR0H:TMR0L=0108H=264(十进制)，65 536−264=65 272。而65 272×64×0.4 μs=1.671 s，或者从例9-12得到65 272×25.6 μs=1.671 s。

例 9-14 假设XTAL=10 MHz。(a)当预分频器值是256时，计算送入定时器0的时钟周期。(b)指出使用预分频器时所能得到的最大时延。

解：

(a) 由于预分频器值为1∶256，所以有1/256×2.5 MHz=9765.625 Hz，1/4×10 MHz=2.5 MHz，并且 T=1/9765.625 Hz=1.024 ms。

(b) 为了得到最大时延，可以把TMR0L和TMR0H都置0。把TMR0H和TMR0L都置0意味着让定时器从0000计数到FFFFH，然后跳变为零，TMR0IF变为高电平。作为结果，定时器历经了65 536个状态。因此，得到的时延=(65 536－0)×1024 μs=67 108 864μs=67.108 864 s。

347

例9-15 假设XTAL=10 MHz，编制程序，在引脚PORTB.7上产生频率为50 Hz的方波。使用定时器0，16位工作模式，使用值为128的预分频器。

解：

请看下面的步骤：

(a) T=1/50 Hz=20 ms，即为方波的周期；

(b) 高低电平的持续时间都等于脉冲的1/2周期，等于10 ms。

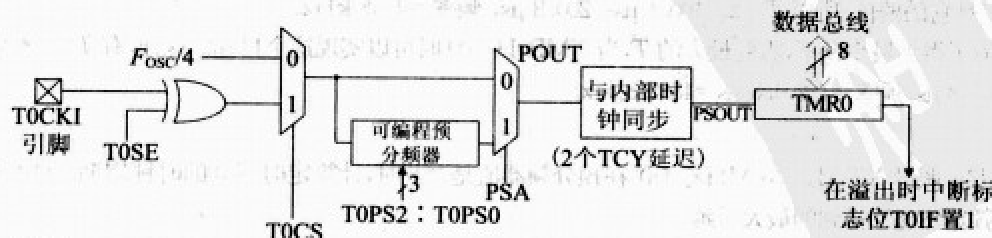
(c) 10 ms/0.4 μs/128=195，65 536－195=65 341(十进制)，十六进制是FF3DH。

(d) 因此TL=3D，TH=FF(十六进制)。

```
        BCF    TRISB,7              ;PB7 as an output
        MOVLW  0x06                 ;Timer0,16-bit,int clk,128 prescale
        MOVWF  T0CON                ;load T0CON reg.
HERE    MOVLW  0xFF                 ;TMR0H = FF, the high byte
        MOVWF  TMR0H                ;load Timer0 high byte
        MOVLW  0x3D                 ;TMR0L = 3DH, the low byte
        MOVWF  TMR0L                ;load Timer0 low byte
        BCF    INTCON, TMR0IF       ;clear timer interrupt flag bit
        BTG    PORTB,7              ;toggle PB7
        BSF    T0CON, TMR0ON        ;start Timer0
AGAIN   BTFSS  INTCON, TMR0IF       ;monitor Timer0 flag until
        BRA    AGAIN                ;it rolls over
        BCF    T0CON, TMR0ON        ;stop Timer0
        BRA    HERE                 ;load TH, TL again
```

9.1.10 定时器0的8位模式编程

定时器0也可以用于8位工作模式。8位模式只允许将00到FFH之间的值放入定时器的寄存器TMR0L。当定时器启动时，TMR0L寄存器执行加法计数，一直计数到上限FFH。当它从FFH跳变为0时，TMR0IF变为高电平。如图9-7所示。



注意：(1) T0CS、T0SE、PSA、T0PS2：T0PS0 (T0CON<5:0>)。

(2) 在复位期间，定时器0被设为8位模式，时钟输入来自T0CK1，预分频器为最大值。

348

图9-7 定时器0的8位模式方框图

9.1.11 定时器 0 的 8 位模式编程步骤

为了使用定时器的 8 位模式来产生时延,需要遵循下面的步骤。

(1) 对 T0CON 寄存器赋值,说明使用的是 8 位工作模式。

(2) 对 TMR0L 赋计数初值;

(3) 启动定时器。

(4) 保持监视定时器标志位(TMR0IF),观察它是否变为高电平。如果 TMR0IF 变为高电平,跳出循环。

(5) 使用指令 BCF T0CON, TMR0ON 停止定时器。

(6) 为执行下一轮循环,清除 TMR0IF 标志位。

(7) 返回到步骤(2),对 TMR0L 重新赋值。

注意,当选择 8 位工作模式时,只会用到 TMR0L 寄存器,而 TMR0H 在计数过程中为 0。为了清楚了解上面步骤的使用,请参阅例 9-16 和例 9-17。

例 9-16 假设 XTAL=10 MHz。计算下面程序中在引脚 PORTB.0 上产生的方波频率,以及本程序能产生方波的最小频率和实现该频率的 TH 值。

```
        BCF    TRISB,0           ;PB0 as an output
        MOVLW  0x48              ;Timer0,8-bit,int clk,no prescaler
        MOVWF  T0CON             ;load T0CON reg.
        BCF    INTCON,TMR0IF     ;clear timer interrupt flag bit
HERE    MOVLW  0x5               ;TMR0L = 5, the low byte
        MOVWF  TMR0L             ;load Timer0 byte
        CALL   DELAY
        BTG    PORTB,0           ;toggle PB0
        BRA    HERE              ;load TL again
;----------delay using Timer0
DELAY   BSF    T0CON,TMR0ON      ;start Timer0
AGAIN   BTFSS  INTCON,TMR0IF     ;monitor Timer0 flag until
        BRA    AGAIN             ;it rolls over
        BCF    T0CON,TMR0ON      ;stop Timer0
        BCF    INTCON,TMR0IF     ;clear Timer0 interrupt flag bit
        RETURN
```

解:

(a) 现在有(256−05)=251×0.4 μs=100.4 μs,即脉冲的高电平部分。因为是一个占空比为 50%的方波,周期 T 是它的两倍;所以 T=2×100.4 μs=200.8 μs, 频率=4.98 kHz。

(b) 为了得到最小频率,需要最大的 T,当 TMR0H=00 时可以实现这个目标。这样,有 T=2×256×0.4 μs=204.8 μs,频率=1/204.8 μs=4882.8 Hz。

例 9-17 假设 XTAL=10 MHz。(a) 在预分频器值是 256 时,计算定时器 0 的时钟周期。(b) 指出这个预分频器下能够得到的最大时延。

解:

(a) 由于预分频器值为 1∶256,所以有 1/256×2.5 MHz=9765.625 Hz,1/4×10 MHz=2.5 MHz,并且 T=1/9765.625 Hz=1024 μs。

(b) 为了得到最大时延,需要置 TMR0L=0。TMR0L 为 0 意味着定时器从 00 数到 FFH,然后跳变为零,TMR0IF 标志位变为高电平。所以,它历经 256 个状态。于是,得到的时延=(256−0)×1024 μs=262 144 μs=0.262 144 s。

9.1.12 编译器和负值

因为定时器是 8 位模式,所以可以让编译器来计算 TMR0H 的值。例如,对于指令 MOVLW, − D'100',编译器会计算−100=9C,然后让 WREG=9C。这样工作变得简单多了。请参阅例 9-18 和例 9-19。

例 9-18 假设定时器工作在 8 位模式,对于下面的各种情况,计算各 TMR0L 的值。

(a)
```
MOVLW  -D'200'
MOVWF  TMR0L
```
(b)
```
MOVLW  -D'60'
MOVWF  TMR0L
```
(c)
```
MOVLW  -D'12'
MOVWF  TMR0L
```

解:

可以使用 Windows 的科学计算器来验证编译器提供的值。在 Windows 的科学计算器下,选择十进制模式,输入 200。然后选择十六进制,再用"+/−"得到负值。下面是计算得到的结果。

十进制	补码(TMR0L 值)
−200	38H
−60	C4H
−12	F4H

350

例 9-19 计算下面代码产生的方波频率和方波的占空比。假设 XTAL=10 MHz。

```
      BCF    TRISB,3           ;PB3 as an output
      BCF    INTCON,TMR0IF     ;clear timer interrupt flag bit
      MOVLW  0x48              ;Timer0,8-bit,int clk,no prescaler
      MOVWF  T0CON             ;load T0CON reg.
HERE  MOVLW  -D'150'           ;loading negative value
      MOVWF  TMR0L             ;load Timer0 byte
      BSF    PORTB,3           ;PB3 = 1
      CALL   DELAY
      MOVWF  TMR0L             ;reload Timer0 byte
      CALL   DELAY
      BCF    PORTB,3           ;PB3 = 0
      MOVWF  TMR0L             ;reload Timer0 byte
      CALL   DELAY
      BRA    HERE              ;load TH, TL again
;----------delay using Timer0
DELAY BSF T0CON,TMR0ON         ;start Timer0
AGAIN BTFSS INTCON,TMR0IF      ;monitor Timer0 flag until
      BRA    AGAIN             ;it rolls over
      BCF    T0CON,TMR0ON      ;stop Timer0
      BCF    INTCON,TMR0IF     ;clear timer interrupt flag bit
      RETURN
```

解：

因为TMR0L工作在8位模式，只要输入为负数，编译器就会自动转换。这使得计算非常方便。因为使用了150个时钟，所以子例程的时延时间=150×0.4 μs=60 μs。脉冲高电平的时间是低电平的两倍（占空比为66%）。因此，得到：T=高电平部分+低电平部分=2×60 μs+60 μs=180 μs，频率=5.555 555 kHz。

这个程序也可以写成下面的形式：

```
        BCF    TRISB,3          ;PB3 as an output
        BCF    INTCON,TMR0IF    ;clear timer interrupt flag bit
        MOVLW  0x48             ;Timer0,8-bit,int clk,no prescaler
        MOVWF  T0CON            ;load T0CON reg.
HERE    BSF    PORTB,3          ;PB3 = 1
        CALL   DELAY
        CALL   DELAY
        BCF    PORTB,3          ;PB3 = 0
        CALL   DELAY
        BRA    HERE             ;load TH, TL again
;————-delay using Timer0
DELAY   MOVLW  -D'150'          ;loading negative value
        MOVWF  TMR0L            ;load Timer0 byte
        BSF    T0CON,TMR0ON     ;start Timer0
AGAIN   BTFSS  INTCON,TMR0IF    ;monitor Timer0 flag until
        BRA    AGAIN            ;it rolls over
        BCF    T0CON,TMR0ON     ;stop Timer0
        BCF    INTCON,TMR0IF    ;clear timer interrupt flag bit
        RETURN
```

351

9.1.13　定时器1编程

定时器1是一个16位定时器，它的16位寄存器分别记作TMR1L（定时器1的低字节）和TMR1H（定时器1的高字节），如图9-8所示。定时器1只能工作在16位模式，和定时器0不同的是，它不支持8位模式。除了TMR1IF（定时器1中断标志）外，定时器1也有T1CON（定时器1控制）寄存器。TMR1IF标志位在TMR1H∶TMR1L从FFFFH到0000溢出时候会变为高电平。定时器1也有预分频器选项，不过它只支持分频系数1∶1、1∶2、1∶4和1∶8。图9-9是定时器1的方框图，而图9-10是T1CON寄存器。PIR1寄存器里包括有TMR1IF标志位，如图9-11所示。

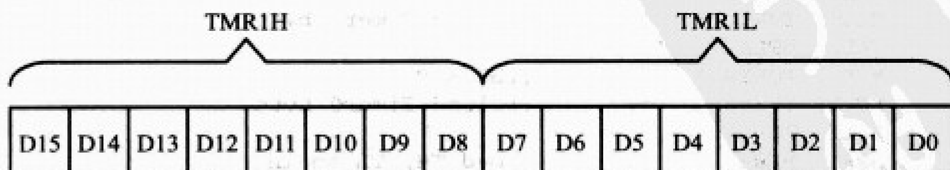


图9-8　定时器1的高低寄存器

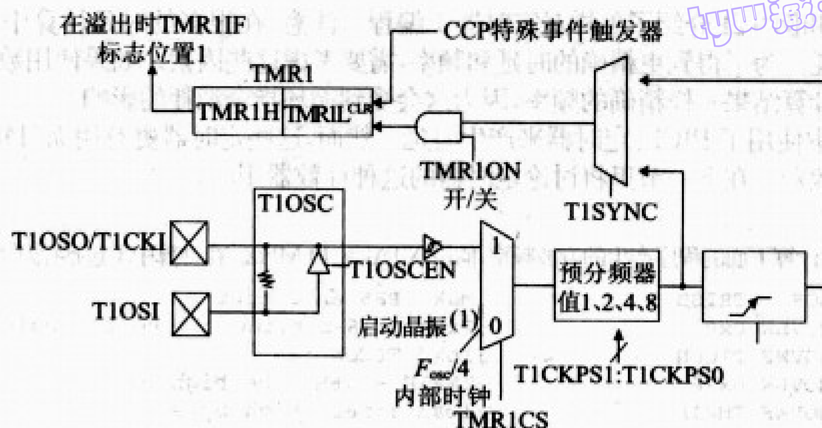


图 9-9 定时器 1 方框图

RD16	---	T1CKPS1	T1CKPS0	T1OSCEN	T1SYNC	TMR1CS	TMR1ON

RD16 D7 16 位读/写使能位
1=定时器 1 的 16 位可按一个 16 位来操作
0=定时器 1 的 16 位可按两个 8 位来操作
D6 未使用

T1CKPS2：T1CKPS0 D5 D4 定时器 1 的预分频器值选择
00=1∶1 预分频器值
01=1∶2 预分频器值
10=1∶4 预分频器值
11=1∶8 预分频器值

T1OSCEN D3 定时器 1 的晶振使能位
1=使能定时器 1 的晶振
0=关闭定时器 1 的晶振

T1SYNC D2 定时器 1 同步(仅当 TMR1CS=1 计数器模式时，与外部时钟输入同步)
如果 TMR1CS=0，该位不使用

TMR1CS D1 定时器 1 的时钟源选择位
1=外部时钟，来自 RC0/T0CKI 引脚
0=内部时钟，来自 XTAL 晶振的 $F_{osc}/4$

TMR1ON D0 定时器 1 的开关控制位
1=启动计数器 1
0=关闭计数器 1

图 9-10 T1CON(定时器 1 控制)寄存器

							TMR1IF

TMR1IF D1 定时器 1 中断溢出标志位
0=定时器 1 无溢出
1=定时器 1 溢出(从 FFFF 到 0000)

TMR1IF 的重要性：当 TMR1H：TMR1L 从 FFFF 到 0000 溢出时，该位就会变为高电平。在重写 TMR1H：TMR1L 寄存器前，要检查该标志位
该寄存器的其他位将在第 11 章中讨论

图 9-11 PIR1(中断控制寄存器 1)包含 TMR1IF 标志

例9-20和例9-21介绍了怎样对定时器1编程。注意，在很多的时延计算中，都忽略了循环的指令消耗。为了得到更精确的时延和频率，需要考虑这些因素。如果使用数字示波器，就不会得到和计算结果一样精确的频率，因为这会受到循环指令消耗的影响。

352 ~ 353

在本节中使用了PIC18定时器来产生时延。然而，这些定时器更有用而且特别的操作是用作事件计数器。在下一节里将讨论定时器的这种计数器用法。

例9-20 计算下面的程序产生的方波频率，假设XTAL=10 MHz。在计算中不包括循环的指令消耗。

```
      BCF    TRISB,5            ;make PB5 an output
      MOVLW  0x0                ;Timer1,16-bit,int clk,no prescale
      MOVWF  T1CON              ;load TOCON reg
HERE  MOVLW  0x76               ;TMR1H = 76H, the high byte
      MOVWF  TMR1H              ;load Timer1 high byte
      MOVLW  0x34               ;TMR1L = 34H, the low byte
      MOVWF  TMR1L              ;load Timer1 low byte
      BCF    PIR1,TMR1IF        ;clear timer interrupt flag bit
      CALL   DELAY
      BTG    PORTB,RB5          ;toggle PB5
      BRA    HERE               ;load TH, TL again
;————————delay using Timer1
DELAY BSF    T1CON,TMR1ON       ;start Timer1
AGAIN BTFSS  PIR1,TMR1IF        ;monitor Timer1 flag until
      BRA    AGAIN              ;it rolls over
      BCF    PIR1,TMR1ON        ;stop Timer1
      RETURN
```

解：

因为FFFFH−7634H=89CBH+1=89CCH，而89CCH=35 276个时钟，35 276×0.4 μs=14.11 ms，频率=1/(14.11 ms×2)=35.434 Hz。这里忽略了用于循环的指令消耗。计算过程和例9-7一样。

例9-21 假设XTAL=10 MHz，编制程序，在引脚PORTB.5上产生频率为50 Hz的方波。使用定时器1，16位模式，使用最大的预分频器值。

解：

因为FFFFH−F3CBH=C34H+1=C35H，C35H=3125个时钟，3125×8×0.4 μs=10 ms，频率=1/(2×10 ms)=50 Hz。计算中忽略了用于循环的指令消耗。

```
      BCF    TRISB,5            ;make PB5 an output
      MOVLW  0x30               ;Timer1,16-bit,int clk,prescale 1:8
      MOVWF  T1CON              ;load T1CON reg
HERE  MOVLW  0xF3               ;TMR1H = F3H, the high byte
      MOVWF  TMR1H              ;load Timer1 high byte
      MOVLW  0xCB               ;TMR1L = CBH, the low byte
      MOVWF  TMR1L              ;load Timer1 low byte
      BCF    PIR1,TMR1IF        ;clear timer interrupt flag bit
      CALL   DELAY
      BTG    PORTB,RB5          ;toggle PB5
      BRA    HERE               ;load TH, TL again
;————————delay using Timer1
DELAY BSF    T1CON,TMR1ON       ;start Timer1
AGAIN BTFSS  PIR1,TMR1IF        ;monitor Timer1 flag until
      BRA    AGAIN              ;it rolls over
      BCF    PIR1,TMR1ON        ;stop Timer1
      RETURN
```

354

9.1.14 复习题

1. PIC18F458/4580 里有多少个定时器?
2. 判断对错:定时器 0 只能用作 16 位定时器。
3. 判断对错:定时器 1 只能用作 16 位定时器。
4. 判断对错:T0CON 是位可寻址的寄存器。
5. 指出指令 MOV T0CON, 0x08 所做的选择。
6. 在 16 位模式下,当计数器从________变成________时,计数器复零。
7. 在 8 位模式下,当计数器从________变成________时,计数器复零。
8. 对于指令 MOVLW- D'200',找出 WREG 里的十六进制数。
9. 为了得到 2 ms 的时延,在 16 位模式下对 TMR0H 和 TMR0L 应该送入什么值。假设 XTAL=10 MHz。
10. 为了得到 100 μs 的时延,在 8 位模式下对 TMR0H 和 TMR0L 应该送入什么值。假设 XTAL=10 MHz。

9.2 计数器编程

在上一节中使用了 PIC18 的定时器来产生时延。然而,这些定时器也可以用来作为外部事件的计数器。本节将讨论怎样把定时器用作事件计数器。当定时器用作定时器时,PIC18 的晶振用作频率源。可是,当定时器用作计数器时,是来自 PIC18 的外部脉冲让 TH 和 TL 加法计数。在计数器模式下,寄存器 T0CON、TMR0H 和 TMR0L 是和上一节中讨论的用于定时器一样的,甚至包括名字也一样。

9.2.1 T0CON 寄存器中的 T0CS 位

回顾上一节,T0CON 寄存器的 T0CS(定时器 0 时钟源)位用来确定定时器的时钟源。当 T0CS=0 时,定时器从引脚 OSC1 和 OSC2 的晶振得到脉冲。相反,当 T0CS=1 时,定时器用作计数器,脉冲来自 PIC18 的外部。因此,当 T0CS=1 时,计数器根据引脚 RA4(PORTA.4)上的脉冲计数。该引脚被称作 T0CKI (定时器 0 的时钟输入)。注意,该引脚属于 PORTA。对于定时器 0,当 T0CS=1 时,引脚 RA4(PORTA.4)提供时钟脉冲,计数器将对该引脚上的每个时钟脉冲计数加 1。相似地,对于计数器 1,当 TMR1CS=1 时,计数器将对引脚 RC0 (PORTC.0)上的每个时钟脉冲计数加 1。请参阅例 9-22。

在例 9-23 中,使用的是计数器 1 作为时间计数器,来计数送入引脚 3.5 的时钟脉冲。这些时钟脉冲可以代表通过一个入口的人数,或者是车轮的转数,或者是其他可以转换成脉冲的事件。

在例 9-23 中,TL 数据以二进制的形式显示。在例 9-24 中,TL 寄存器的值先被转换为 ASCII 码,然后在 LCD 上显示。

另一个使用定时器 C/T=1 的例子是把频率为 60 Hz 的外部方波送入定时器。程序会根据输入频率产生秒、分钟和小时,并把结果显示在 LCD 上。这就是一个很好的电子钟,只是不太精确而已。

例 9-22　计算 T0CON 的值。假设定时器 0 工作在 8 位模式,不使用预分频器。另外,使用外部时钟作为时钟源,在上升沿增量计数。

解:

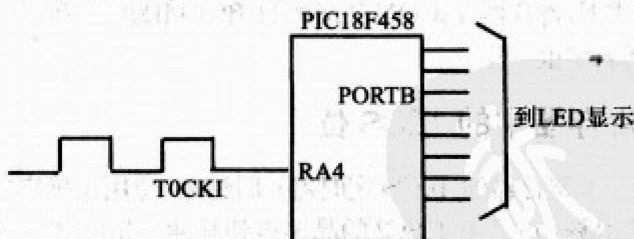
T0CON=01101000　即 8 位模式,外部时钟源,不使用预分频器。

例 9-23　假设时钟脉冲输入至引脚 T0CKI。编制程序,让计数器 0 工作在 8 位模式来计算脉冲数,然后把 TMR0L 的状态显示在 PORTB。

解:

```
       BSF    TRISA,RA4          ;PORTA.4 as an input for clock
       CLRF   TRISB              ;PORTB as an output
       MOVLW  0x68               ;Timer0, 8-bit,ext clk,no prescale
       MOVWF  T0CON              ;load T0CON reg
HERE   MOVLW  0x0                ;TMR0L = 0
       MOVWF  TMR0L              ;load Timer0
       BCF    INTCON,TMR0IF      ;clear timer interrupt flag bit
       BSF    T0CON,TMR0ON       ;start Timer0
AGAIN  MOVFF  TMR0L,PORTB        ;display the count on PORTB
       BTFSS  INTCON,TMR0IF      ;monitor Timer0 flag until
       BRA    AGAIN              ;it rolls over
       BCF    T0CON,TMR0ON       ;stop Timer0
       GOTO   HERE
```

PORTB 连接到 8 个 LED 并输入 T0CKI 到脉冲。



9.2.2　使用外部晶振作为定时器 1 的时钟

当定时器 1 使用外部时钟源时,有两个选择。它可以使用引脚 T1CKI 上的时钟,或者是 T1OSI 和 T1OSO 引脚的晶振时钟,如图 9-9 所示。通常,将一个 32 kHz 的晶振连接到 T1OSI 和 T1OSO 引脚,用于在休眠模式下节省电源,因为休眠指令不会关闭定时器 1。注意,这个连接到 T1OSI 和 T1OSO 引脚的 32 kHz 晶振,不同于连接至引脚 OSC1 和 OSC2 的主晶振。PIC18 使用主晶振来执行 CPU 指令时钟周期,而当 CPU 进入休眠模式后,主晶振将关闭以节省电源。在休眠模式下,主晶振被关闭,替代的是 T1OSI 和 T1OSO 引脚上的 32 kHz 晶振为定时器 1 提供时钟。这就可以使用定时器来实现片上 RTC(实时时钟)。第 16 章将介绍如何将外部 RTC 连接到 PIC18 中。注意,为了使用定时器 1 的外部时钟源,必须使能 T1CON 寄存器的 T1OSCEN 位(T1OSCEN=1),同时还要选择外部时钟源选项 TMR1CS=1,如图 9-10

所示。学习例 9-23～例9-27，观察定时器是如何用作计数器的。

在本节结束前，需要指出重要的一点。你可能会觉得监视 TMR0IF 和 TMR1IF 标志位是浪费微控制器的时间。你是对的。一个解决办法是使用中断。使用中断可以让微控制器做其他的事情。当一个定时器的中断标志位（如 TMR0IF）变为高电平时，它将会发出通知。关于 PIC18 的这个重要而强大的功能将在第 11 章中讨论。

357

例 9-24 假设将 1 Hz 的脉冲信号连接到定时器 0 的输入（引脚为 T0CKI）。编制程序，在 PORTB、PORTC 和 PORTD 上显示计数器 0 的十进制数。设 TMR0L 的初值为－60。

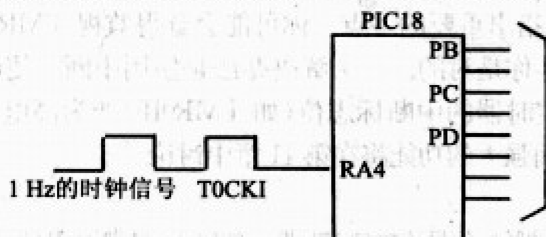
解：

要在 LCD 上显示 TMR0L 计数，必须把 8 位二进制数转换为 ASCII 码。关于数据转换，请参阅第 5 章。

```
NUME        EQU   0x00        ;RAM loc for NUME
QU          EQU   0x20        ;RAM loc for quotient
RMND_L      EQU   0x30        ;the least significant digit loc
RMND_M      EQU   0x31        ;the middle significant digit loc
RMND_H      EQU   0x32        ;the most significant digit loc
MYDEN       EQU   D'10'       ;value for divide by 10
      BSF   TRISA,RA4         ;RA4 as an input
      MOVLW 0x68              ;Timer0,8-bit, ext clk,no prescale
      MOVWF T0CON             ;load T0CON reg
HERE  MOVLW 0x0               ;TMR0L = 0
      MOVWF TMR0L             ;load Timer0
      BCF   INTCON,TMR0IF     ;clear timer interrupt flag bit
      BSF   T0CON,TMR0ON      ;start Timer0
AGAIN MOVF  TMR0L,W           ;save the count in WREG
      CALL  BIN_ASC_CON
      BTFSS INTCON,TMR0IF     ;monitor Timer0 flag until
      BRA   AGAIN             ;it rolls over
      BCF   T0CON,TMR0ON      ;stop Timer0
      GOTO  HERE

;converting 8-bit binary to decimal
BIN_DEC_CON
      MOVFF PORTB,WREG
      MOVWF NUME              ;load numerator
      MOVLW MYDEN             ;WREG = 10, the denominator
      CLRF  QU                ;clear quotient
D_1   INCF  QU                ;inc quotient for every subtract
      SUBWF NUME              ;subtract WREG from NUME value
      BC    D_1               ;if positive go back
      ADDWF NUME              ;once too many, first digit
      DECF  QU                ;once too many for quotient
      MOVFF NUME,RMND_L       ;save the first digit
      MOVFF QU,NUME           ;repeat the process one more time
      CLRF  QU                ;clear QU
D_2   INCF  QU
      SUBWF NUME              ;subtract WREG from NUME value
      BC    D_2
      ADDWF NUME              ;once too many
      DECF  QU
      MOVFF NUME,RMND_M       ;2nd digit
      MOVFF QU,RMND_H         ;3rd digit
      RETURN
```

为了在 LCD 中显示数据，必须将十进制数转换成 ASCII 码。请参阅第 6 章。



358

例 9-25　假设将 16 Hz 的脉冲连接到定时器 0 的输入(引脚 T0CKI)。编制程序，在 PORTB 和 PORTD 上显示 TMR0H 和 TMR0 L 的值。初值设为 0。使用计数器 0，16 位模式，使用上升沿时钟。写出下列选择时的程序：(a) 不使用预分频器，(b) 使用预分频器值 1∶16。

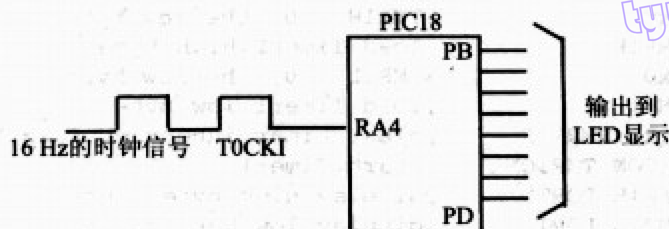
解：

(a)

```
      BSF    TRISA,RA4         ;RA4 as an input
      CLRF   TRISB             ;PORTB as an output
      CLRF   TRISD             ;PORTD as an output
      MOVLW  0x28              ;Timer 0,16-bit,ext clk,no prescale
      MOVWF  T0CON             ;load T0CON reg
HERE  MOVLW  0x0               ;TMR0H = 0
      MOVWF  TMR0H             ;load Timer0 high byte
      MOVLW  0x0               ;TMR0L = 0
      MOVWF  TMR0L             ;load Timer0 low byte
      BCF    INTCON,TMR0IF     ;clear timer interrupt flag bit
      BSF    T0CON,TMR0ON      ;start Timer0
AGAIN MOVFF  TMR0H,PORTD       ;display high byte count
      MOVFF  TMR0L,PORTB       ;display low byte count
      BTFSS  INTCON,TMR0IF     ;monitor Timer0 flag until
      BRA    AGAIN             ;it rolls over
      BCF    T0CON,TMR0ON      ;stop Timer0
      GOTO   HERE
```

(b)

```
      BSF    TRISA,RA4         ;RA4 as an input
      CLRF   TRISB             ;PORTB as an output
      CLRF   TRISD             ;PORTD as an output
      MOVLW  0x23              ;T0,16-bit,ext clk,prescale of 1:16
      MOVWF  T0CON             ;load T0CON reg
HERE  MOVLW  0x0               ;TMR0H = 0
      MOVWF  TMR0H             ;load Timer0 High byte
      MOVLW  0x0               ;TMR0L = 0
      MOVWF  TMR0L             ;load Timer0 low byte
      BCF    INTCON,TMR0IF     ;clear timer interrupt flag bit
      BSF    T0CON,TMR0ON      ;start Timer0
AGAIN MOVFF  TMR0H,PORTD       ;display high byte count
      MOVFF  TMR0L,PORTB       ;display low byte count
      BTFSS  INTCON,TMR0IF     ;monitor Timer0 flag until
      BRA    AGAIN             ;it rolls over
      BCF    T0CON,TMR0ON      ;stop Timer0
      GOTO   HERE
```



359

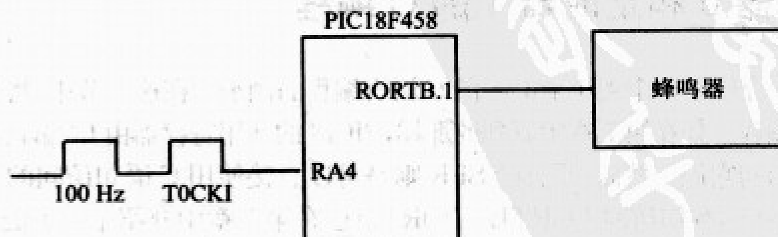
例 9-26 假设将时钟脉冲输入到引脚 T0CKI,将蜂鸣器连接到引脚 PORTB.1。编制程序,使用计数器 0 的 8 位工作模式,让蜂鸣器每隔 100 个脉冲发声一次。

解:

为了让蜂鸣器每隔 100 个脉冲发声一次,需要把计数器初值设为－100(9CH),然后启动计数器开始计数,直到 FFH。当溢出时,可以通过翻转 PORTB.1 引脚来使蜂鸣器发声。

```
      BCF    TRISB,1          ;RB1 as an output
      BSF    TRISA,4          ;RA4 as an input for clock-in
      MOVLW 0x68              ;Timer0,8-bit,ext clk,no prescale
      MOVWF T0CON             ;load T0CON reg
      MOVLW -D'100'           ;TMR0L = 0
      MOVWF TMR0L             ;load Timer0
      BCF    INTCON,TMR0IF    ;clear timer interrupt flag bit
      BSF    T0CON,TMR0ON     ;start Timer0
AGAIN BTFSS INTCON,TMR0IF     ;monitor Timer0 flag until
      BRA    AGAIN            ;it rolls over
      BCF    T0CON,TMR0ON     ;stop Timer0
OVER  BTG    PORTB,1          ;sound the buzzer
      CALL   DELAY            ;quarter second delay
      GOTO   OVER             ;forever
```

将 PORTB 的第 1 位连接到蜂鸣器,输入为 T0CKI 脉冲。



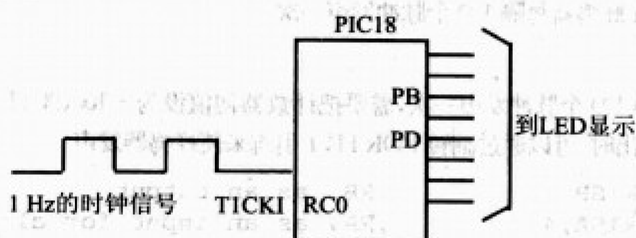
360

例 9-27 假设将 1 Hz 的脉冲连接到定时器 1 的输入(引脚 PORTC.0)。编制程序,在 PORTB 和 PORTD 上显示 TMR1H 和 TMR1L 的值。初值设为 0。使用计数器 1,16 位模式,上升沿时钟。

解:

```
BSF   TRISC,RC0      ;PC0 as an input
CLRF  TRISB          ;PORTB as an output
CLRF  TRISD          ;PORTD as an output
MOVLW 0x02           ;Timer1,16-bit,ext clk,no prescale
MOVWF T1CON          ;load T0CON reg
```

```
HERE  MOVLW 0x0          ;TMR1H = 0, the low byte
      MOVWF TMR1H        ;load Timer1 high byte
      MOVLW 0x0          ;TMR1L = 0, the low byte
      MOVWF TMR1L        ;load Timer1 low byte
      BCF   PIR1,TMR1IF  ;clear timer interrupt flag bit
      BSF   T1CON,TMR1ON ;start Timer1
AGAIN MOVFF TMR1H,PORTD  ;display high byte count
      MOVFF TMR1L,PORTB  ;display low byte count
      BTFSS PIR1,TMR1IF  ;monitor Timer1 flag until
      BRA   AGAIN        ;it rolls over
      BCF   PIR1,TMR1ON  ;stop Timer1
      GOTO  HERE
```



9.2.3 复习题

1. 当 T0CS=0 时,由什么向 PIC18 定时器提供时钟脉冲?
2. 当 T0CS=1 时,由什么向 PIC18 定时器提供时钟脉冲?
3. 如果 T0CS=1,那么在 9.1 节中所讨论的是否还适用于定时器?
4. 要使 RC0 作为定时器 1 时钟的输入端口,需要做些什么设置? 请给出理由。
5. 程序员可以选择计数器是上升沿计数还是下降沿计数吗?

9.3 定时器 0 和定时器 1 的 C 编程

在第 7 章中介绍了几个关于 PIC18 的 C 语言编程的例子。在这一节中,将介绍 PIC18 定时器的 C 语言编程。如在第 7 章中看到的那样,PIC18 的通用寄存器由 C 编译器控制,不能使用 C 语句直接访问它们。然而,所有的 SFR 则是可以直接使用 C 语句访问的。作为一个直接访问 SFR 的例子,如何访问 PORTB~PORTD 已在第 7 章中介绍了。下面将讨论在 C18 编译器下如何直接访问 PIC18 定时器。

9.3.1 用 C 访问定时器

在 C18 里,可以使用 PIC18Fxxx.h 头文件直接访问定时器寄存器,如 TMR0H、TMR0L 和 T0CON。例 9-28 和例 9-29 说明了怎样访问 TMR0ON 和 TMR0IF 标志位。注意,所有的 SFR 寄存器都是可位寻址的。

9.3.2 计算使用定时器的时延

在上两节中已看到,时延时间取决于两个因素:晶振频率和预分频器值。影响时延大小

的第3个因素是C编译器,因为不同的C编译器会产生不同的十六进制代码长度。研究例9-28~例9-33,使用示波器验证它们。

362

例9-28 编制C18程序,带时延地不断翻转PORTB的所有位。使用计数器0,16位模式,而不使用预分频器来产生时延。

解:

```
#include <p18f4580.h>
void T0Delay(void);
void main(void)
  {
  TRISB=0;                    //PORTB output port
  while(1)                    //repeat forever
    {
      PORTB=0x55;             //toggle all bits of Port B
      T0Delay();              //delay size unknown
      PORTB=0xAA;             //toggle all bits of Port B
      T0Delay();
    }
  }

void T0Delay()
  {
    T0CON=0x08;               //Timer0, 16-bit mode, no prescaler
    TMR0H=0x35;               //load TH0
    TMR0L=0x00;               //load TL0
    T0CONbits.TMR0ON=1;             //turn on T0
    while(INTCONbits.TMR0IF==0);    //wait for TF0 to roll over
    T0CONbits.TMR0ON=0;             //turn off T0
    INTCONbits.TMR0IF=0;      //clear TF0
  }
```

363

例9-29 编制C18程序,每隔50 ms翻转PORTB 4位。使用计数器0,16位模式,使用1:4的预分频器值来产生时延。假设XTAL=10 MHz。

解:

```
#include <p18f4580.h>
void T0Delay(void);
#define mybit PORTBbits.RB4
void main(void)
  {
   TRISBbits.TRISB4=0;
   while(1)
    {
      mybit^=1;                    //toggle PORTB.4
      T0Delay();                   //Timer0, mode 1 (16-bit)
    }
  }
```

```
void T0Delay()
   {
     T0CON=0x01;                     //Timer0, 16-bit mode, 1:4 prescaler
     TMR0H=0x85;                            //load TH0
     TMR0L=0xEE;                            //load TL0
     T0CONbits.TMR0ON=1;                    //turn on Timer0
     while(INTCONbits.TMR0IF==0);           //wait for TF0 to roll over
     T0CONbits.TMR0ON=0;                    //turn off Timer0
     INTCONbits.TMR0IF=0;                   //clear TF0
   }
```

FFFFH−85EEH=7A11H≐31 249+1=31 250

364

时延=31 250×4×0.4 μs=50 ms

例 9-30 编制 C18 程序，在引脚 PORTB.5 上产生 2 Hz 的频率信号。使用计数器 0、8 位模式来产生时延。

解：

```
#include <p18f4580.h>
void T0M8Delay(void);
#define mybit PORTBbits.RB5
void main(void)
  {
     unsigned char x,y;
    TRISBbits.TRISB5 = 0;
    while(1)
      {
        mybit^=1;                  //toggle PortB.5
        for(x=0;x<250;x++)         //due to for loop overhead
          for(y=0;y<35;y++)        //we put 35 and not 39
            T0M8Delay();
      }
  }
void T0M8Delay()
  {
    T0CON=0x45;         //Timer0, 16-bit mode, prescaler 1:64
    TMR0L=-1;                      //load TL0
    T0CONbits.TMR0ON=1;            //turn on T0
    while(INTCONbits.TMR0IF==0);   //wait for TF0 to roll over
    T0CONbits.TMR0ON=0;            //turn off T0
    INTCONbits.TMR0IF=0;           //clear TF0
  }
```

256−255=1

1×64×0.4 μs=25.6 μs

通过计算，有 25.6 μs×250×39=0.2496

$F=1/(2\times0.2496\ \text{s})=1/0.4992\ \text{s}=2\ \text{Hz}$

然而，这并不是示波器的输出结果。这是由 C 程序中循环指令引起的。为了纠正这个问题，可以使用 35

365

代替 39。

例9-31 编制C18程序，在PORTC的所有位上产生250 Hz的频率信号。使用计数器0、16位模式，而不使用预分频器来产生频率。假设XTAL=10 MHz。

解：

```
#include <p18f4580.h>
void T0Delay(void);
void main(void)
  {
    unsigned char x;
    TRISC=0;                  //PORTC output port
    PORTC=0x55;
    while(1)
      {
        PORTC=~PORTC;         //toggle all bits of Port C
        for(x=0;x<20;x++)
          T0Delay();
      }
  }

void T0Delay()
  {
    T0CON=0x0;          //Timer 0, 16-bit mode, no prescaler
    TMR0H=0xFF;         //load TH0
    TMR0L=0x06;         //load TL0
    T0CONbits.TMR0ON=1;                //turn on T0
    while(INTCONbits.TMR0IF==0); //wait for TF0 to roll over
    T0CONbits.TMR0ON=0;                //turn off T0
    INTCONbits.TMR0IF=0;               //clear TF0
  }
```

FF06H=65 286(十进制)

65 536−65 286=250

250×0.4 μs=0.1 ms，且20×0.1 ms=2 ms

T=1/(2×2 ms)=1/4 ms=250 Hz

另一种计算方法是：

T=1/250 Hz=0.004 s，半周期是0.002 s

0.002 s/0.4 μs=5000

因为循环次数为20，所以有5000/20=250。

366

例9-32 将开关连接到PORTB.7。编制C18程序，监视SW的状态，并且在引脚PORTB.0上产生下面频率的时钟信号：

SW=0：500 Hz

SW=1：750 Hz

使用计数器0和预分频器来产生时钟信号。

解：

```
#include <p18f4580.h>
#define mybit PORTBbits.RB0
#define SW PORTBbits.RB7
void T0PSDelay(unsigned char);
```

```
void main(void)
  {
     TRISBbits.TRISB7=1;//make PB.7 an input
      TRISBbits.TRISB0=0;//make PB.0 an output
     SW=1;
     while(1)
       {
        mybit^=1;       //toggle PB.0
        if(SW==0)       //check switch
          T0PSDelay(0);
        else
          T0PSDelay(1);
       }
  }
void T0PSDelay(unsigned char c)
  {
     T0CON=0x05;          //Timer 0, 16-bit mode, prescaler 1:64
     if(c==0)
       {
        TMR0H=0xFF;             //load TH0
        TMR0L=0xD9;             //load TL0
       }
     else
       {
        TMR0H=0xFF;             //load TH0
        TMR0L=0xE6;             //load TL0
       }
     T0CONbits.TMR0ON=1;              //turn on T0
     while(INTCONbits.TMR0IF==0);     //wait for TF0 to roll over
     T0CONbits.TMR0ON=0;              //turn off T0
     INTCONbits.TMR0IF=0;             //clear TF0
  }
```

FFD9H=65 497	FFE6H=65 510
65 536−65 497=39	65 536−65 510=26
39×64×0.4 μs=998 μs	26×64×0.4 μs=666 μs
1/(998 μs×2)=501 Hz	1/(666 μs×2)=751 Hz

367 使用示波器并修改 TH：TL 的值，以获得准确的时钟频率。

例 9-33 编制 C18 程序，在 PORTB.1 位产生 2500 Hz 的频率。使用计数器 1 来生成时延。

解：

```
#include <p18f4580.h>
void T1Delay(void);
#define mybit PORTBbits.RB1

void main(void)
  {
     TRISBbits.TRISB1 = 0;
    while(1)
      {
       mybit^=1;          //toggle PB.1
```

```
            T1Delay();
        }
    }

void T1Delay()
    {
      T1CON=0x0;               //Timer1, 16-bit mode, no prescaler
      TMR1H=0xFE;              //load TH1
      TMR1L=0x0C;              //load TL1
      T1CONbits.TMR1ON=1;              //turn on T1
      while(PIR1bits.TMR1IF==0);   //wait for TF1 to roll over
      T1CONbits.TMR1ON=0;              //turn off T1
      PIR1bits.TMR1IF=0;               //clear TF1
    }
```

1/2500 Hz=400 μs

400 μs/2=200 μs

200 μs/0.4 μs=500

65 536−500=65 036=FE0CH

9.3.3 定时器0和定时器1用作计数器的C编程

在9.2节介绍了把定时器0和定时器1用作事件计数器的情况。当使用外部脉冲代替晶振频率作为时钟源时，定时器就可以用作计数器。将脉冲输入到T0CKI(RA4)和T1CKI(PC0)引脚，就等于把定时器0和定时器1变成了计数器0和计数器1。通过例9-34～例9-37，研究如何使用C语言把定时器0和1编程为计数器。

368

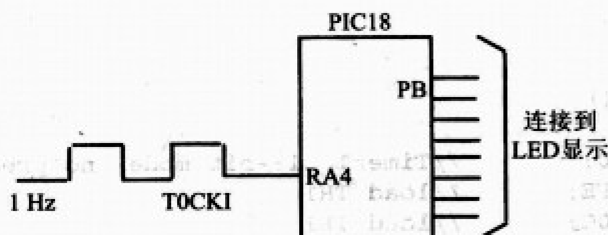
例9-34 假设将1 Hz的外部时钟输入到引脚T0CKI(RA4)。编制C18程序，让计数器0工作在8位模式，并显示PORTB的状态TMR0L。从0H开始计数。

解：

```
#include <p18f4580.h>

void main(void)
  {
    TRISAbits.TRISA4=1;//make RA4/T0CKI an input
      TRISB=0;
    T0CON=0x68;            //Counter 0, 8-bit mode, no prescaler
    TMR0L=0;                               //set count to 0
    while(1)                               //repeat forever
      {
       do
        {
          T0CONbits.TMR0ON=1;              //turn on T0
          PORTB=TMR0L;                     //place value on pins
        }
        while(INTCONbits.TMR0IF==0);       //wait for TF0 to roll over
        T0CONbits.TMR0ON=0;                //turn off T0
        INTCONbits.TMR0IF=0;               //clear TF0
      }
  }
```

PORTB连接有8个LED,且T0CKI(RA4)连接到1 Hz的外部时钟。



369

例9-35 假设将1 Hz的外部时钟输入到引脚T0CKI(RA4)。编制C18程序,让计数器0工作在模式1(16位)来记录脉冲数,并在PORTD和PORTB上分别显示TMR0H和TMR0L寄存器的内容。

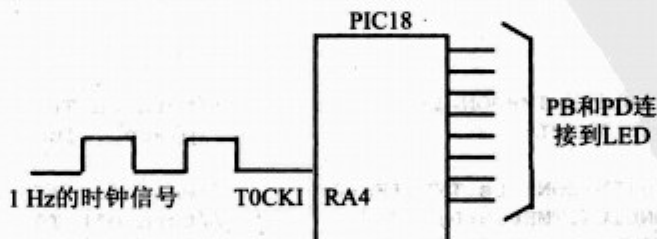
解:

```
#include <p18f4580.h>

void main(void)
  {
    TRISAbits.TRISA4=1;          //make RA4 an input for T0CKI
    TRISB=0;                     //PORTB output port
    TRISD=0;                     //PORTD output port
    T0CON=0x25;       //Timer0, 16-bit mode, prescaler 1:64
    TMR0H=0;                     //set count to 0
    TMR0L=0;                     //set count to 0

    while(1)                     //repeat forever
      {
       do
        {
          T0CONbits.TMR0ON=1;    //turn on T0
          PORTB=TMR0L;           //
          PORTD=TMR0H;           //place value on pins
        }
        while(INTCONbits.TMR0IF==0);//wait for rollover

        T0CONbits.TMR0ON=0;          //turn off T0
        INTCONbits.TMR0IF=0;         //clear TF0
      }
  }
```



例9-36 假设将一个64 Hz的外部时钟输入到引脚T0CKI(RA4)。编制C18程序,让计数器0工作在8位模式来显示ASCII码形式的计数值。8位二进制计数必须要转换成ASCII码,然后分别在PORTB、PORTC和PORTD上显示ASCII码(二进制),其中PORTB是最低位。置TMR0L初值为0。 [370]

解:

为了显示TMR0L计数,必须把8位二进制数转换成ASCII码。关于数据转换,请参阅第7章。ASCII值就显示为二进制数。例如,"9"将在端口显示成"00111001"。

```
#include <p18f4580.h>
void BinToASCII(unsigned char);
void main()
  {
    unsigned char value;
    TRISAbits.TRISA4=1;          //make RA4 an input
     TRISB=0;                    //make PORTB an output
     TRISC=0;                    //make PORTC an output
     TRISD=0;                    //make PORTD an output
     TMR0L=0;
     T0CON=0x65;      //Counter 0, 8-bit mode, prescaler 1:64

    while(1)
      {
       do
        {
          T0CONbits.TMR0ON=1;    //turn on T0
          value=TMR0L;
          BinToASCII(value);
        }
    while(INTCONbits.TMR0IF==0); //wait for TF0 to roll over
    T0CONbits.TMR0ON=0;          //turn off T0
    INTCONbits.TMR0IF=0;         //clear TF0
      }
  }

void BinToASCII(unsigned char value)        //see Chapter 7
  {
    unsigned char x,d1,d2,d3;
    x=value/10;
    d1=value%10;
    d2=x%10;
    d3=x/10;
    PORTB=0x30 | d1;
    PORTC=0x30 | d2;
    PORTD=0x30 | d3;
  }
```

[371]

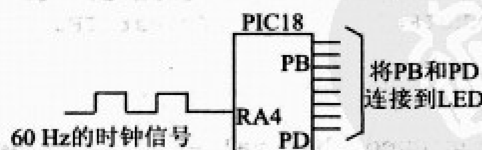
例9-37 假设将一个60 Hz的外部时钟输入到引脚T0CKI(RA4)。编制C18程序,让计数器工作在8位模式,并在PORTB和PORTD上分别显示秒和分。

解：

```
#include <p18f4580.h>

void ToTime(unsigned char);
void main()
  {
    unsigned char sec;
     TRISB=TRISD=0;                     //PORTB,D outputs
    T0CON=0x68;                         //Timer 0, no prescaler
     TMR0L=-60;                         //sec = 60 pulses
    while(1)
      {
       do
        {
          T0CONbits.TMR0ON=1;           //turn on T0
          sec=TMR0L;
          ToTime(sec);
        }
    while(INTCONbits.TMR0IF==0);        //wait for TF0 to roll over
    T0CONbits.TMR0ON=0;                 //turn off T0
    INTCONbits.TMR0IF=0;                //clear TF0
      }
  }

void ToTime(unsigned char value)
  {
    unsigned char sec, min;
    min = value / 60;
    sec = value % 60;
    PORTB = sec;
    PORTD = min;
  }
```



372

使用60 Hz时，可以产生秒、分和小时。

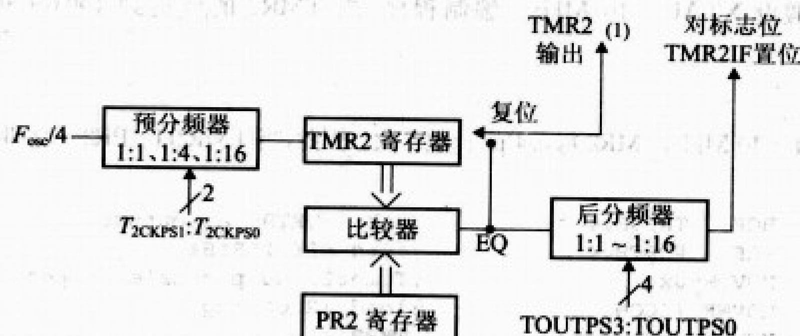
9.4 定时器2和定时器3的编程

本节将介绍PIC18的定时器2和定时器3，以及如何使用汇编语言和C语言对它们进行编程。

9.4.1 定时器2的编程

定时器2是一个8位的定时器。定时器2的8位寄存器被称作TMR2。定时器2还有一个8位的寄存器被称作周期寄存器(PR2)。可以设置PR2为一个固定值，然后定时器2就会从00开始增加，直到同PR2中的值相匹配。此时，两值匹配的信号会让TMR2IF标志位变为高电平，而TMR2则复位为00。定时器2的时钟源是带有可选预分频器和后分频器的$F_{osc}/4$，如图9-12所示。注意，从图9-12可知，定时器2没有外部时钟源。换句话说，它不能作为计数

器使用。下面通过一些例子来学习定时器2的编程语法。请看图9-12~图9-14。



(1)：TMR2寄存器输出可以由SSP模块来软件选择作为波特率时钟。

图 9-12 定时器2的方框图

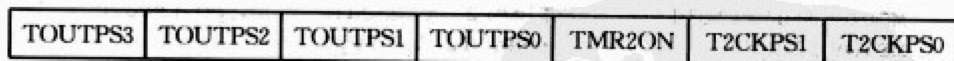


TMR2IF　　定时器2中断溢出标志位
0＝TMR2的值不等于PR2寄存器的内容
1＝TMR2的值等于PR2寄存器的内容

该寄存器的其他位将在第11章中讨论。
TMRxIF在PIR寄存器的位置在更新的产品中可能有变动。

图 9-13 PIR1(外部中断标志寄存器1)包含TMR2IF标志位

373



D7　未使用

TOUTPS3：TOUTPS0　D6～D3　定时器2输出的后分频器值选择位
0000＝1：1　后分频器值
0001＝1：2　后分频器值
0010＝1：3　后分频器值
0011＝1：4　后分频器值
1110＝1：15　后分频器值
1111＝1：16　后分频器值

TMR2ON　D2　定时器2的开关控制位
1＝使能(启动)定时器2
2＝停止定时器2

T2CKPS1：T2CKPS0　D1～D0 定时器2时钟的预分频器值选择位
00＝预分频器值为1
01＝预分频器值为4
1x＝预分频器值为16

图 9-14 T2CON(定时器2控制)寄存器

例 9-38　假设 XTAL＝10 MHz。编制程序，当 TMR2 的值达到 100（十进制）时，打开 PORTB4 引脚。

解：

因为 XTAL=10 MHz，TMR2 每 0.4 μs 计数 1 次。因此，当 TMR2H=PR2=100 时，PORTB4 会打开。

```
        BCF    TRISB,4           ;make PORTB4 an output
        BCF    PORTB,4           ;turn off PORTB4
        MOVLW  0x0               ;Timer2, no prescale or postscale
        MOVWF  T2CON             ;load T2CON reg
        MOVLW  0x0               ;TMR2 = 0
        MOVWF  TMR2              ;load Timer2
        MOVLW  D'100'            ;PR2 = 100, the period register
        MOVWF  PR2               ;load PR2
        BCF    PIR1,TMR2IF       ;clear timer interrupt flag
        BSF    T2CON,TMR2ON      ;start Timer2
AGAIN   BTFSS  PIR1,TMR2IF       ;monitor Timer2 flag
        BRA    AGAIN             ;
        BSF    PORTB,4           ;turn on PORTB4
        BCF    T2CON,TMR2ON      ;stop Timer2
HERE    BRA    HERE
```

374

例 9-39　使用预分频器和后分频器，计算定时器 2 可以产生的最大时延。假设 XTAL=10 MHz。

解：

可以置 PR2=255 来产生最大的时延。当 TMR2 到达值 255(十进制)时，它将翻转某一个引脚。

```
        BCF    TRISB,4           ;make PORTB4 an output
        BCF    PORTB,4           ;turn off PORTB4
        MOVLW  B'01111011'       ;Timer2, prescale = 16,postscale = 16
        MOVWF  T2CON             ;load T2CON reg
        MOVLW  0x0               ;TMR2 = 0
        MOVWF  TMR2              ;load Timer2
HERE    MOVLW  D'255'            ;PR2 = 255, the period register
        MOVWF  PR2               ;load PR2
        BCF    PIR1,TMR2IF       ;clear timer interrupt flag bit
        BSF    T2CON,TMR2ON      ;start Timer2
AGAIN   BTFSS  PIR1,TMR2IF       ;monitor Timer2 flag
        BRA    AGAIN             ;
        BTG    PORTB,4           ;turn on PORTB4
        BCF    T2CON,TMR2ON      ;stop Timer2
        BRA    HERE
```

因为 XTAL=10 MHz，所以 TMR2 每 0.4 μs 计数。当 TMR2H=PR2=255 时，RB4 每隔52 ms在打开和关闭之间切换，因为 255×0.4 μs×16×16=26.112 ms。

例 9-40　假设 XTAL=10 MHz。编制 C18 程序，当 TMR2 的值达到 100(十进制)时，打开 PORTB4 引脚。这是对例 9-13 使用 C18 的一个版本。

解：

```
#include <p18f4580.h>
#define mybit PORTBbits.RB4
void main(void)
  {
    TRISBbits.TRISB4=0;   //PORTB4 as output
  T2CON=0x0;             //Timer2, no prescaler/postscaler
  TMR2=0x00;             //TMR2 = 0
  mybit=0;               //PB.4 = 0
  PR2=100;               //load period register 2
  T2CONbits.TMR2ON=1;    //turn on T0
  while(PIR1bits.TMR2IF==0); //wait for TMR2IF to be raised
  mybit=1;               //PB.4 = 0
  T2CONbits.TMR2ON=0;    //turn off T2
  PIR1bits.TMR2IF=0;     //clear TF0
  while(1);              //stay here
}
```

375

例9-41 使用预分频器和后分频器，计算定时器2可以产生的最大时延。假设XTAL=10 MHz。这是例9-39的C18版本。

解：

```
#include <p18f4580.h>
#define mybit PORTBbits.RB4
void main(void)
  {
    TRISBbits.TRISB4=0;
    T2CON=0x7B;          //Timer2, prescaler = 16,postscaler = 16
    TMR2=0x00;                      //TMR2 = 0
      while(1)
       {
      PR2=255;                      //load period register 2
      T2CONbits.TMR2ON=1;           //turn on T2
      while(PIR1bits.TMR2IF==0);    //wait for TMR2IF to be raised
      mybit=~mybit;                 //toggle PORTB4
      T2CONbits.TMR2ON=0;           //turn off T2
     PIR1bits.TMR2IP=0;             //clear TF0
       }
  }
```

因为XTAL=10 MHz，所以TMR2每0.4 μs计数。当TMR2H=PR2=255时，PORTB4每隔52 ms在打开和关闭之间切换，因为255×0.4 μs×16×16=26.112 ms。

9.4.2 定时器3的编程

定时器3是一个16位的定时器，可以用作定时器或者计数器。它的16位寄存器分成TMR3L和TMR3H两个字节。如图9-17所示，定时器3支持16位模式，但是不支持8位模式。利用T3CON(定时器3控制)寄存器来选择定时器3的各种功能，如图9-15所示。定时器3支持的预分频器值有1∶1、1∶2、1∶4和1∶8，如图9-15所示。图9-15还画出了PIC18的CCP(比较/捕获脉冲宽度调制)特性的相关位。CCP是PIC18很常用的一个特性，将在第15章和第17章中讨论。在第15章中将会看到如何使用CCP特性和中断来测量脉冲宽度。脉

冲宽度调制(PWM)是DC电机控制的一个重要概念,这将在第17章中详细介绍。因为定时器3是一个16位定时器,TMR3IF标志位在TMR3H:TMR3L从FFFFH溢出到0000时变为高电平。TMR3IF(定时器3中断标志位)是PIR2寄存器的一部分,如图9-16所示。

376

研究下面的一些语法,学习定时器3的编程语法。

RD16	T3CCP2	T3CKPS1	T3CKPS0	T3CCP1	T3SYNC	TMR3CS	TMR3ON

RD16 D7 16位读/写使能位
1=定时器3的16位可以按一个16位数来访问
0=定时器3的16位可以按两个8位数来访问

T3CCP2:T3CCP1 D6 D3 定时器3和定时器1的CPPx使能位
00=定时器1是用于CCP模块的比较/捕获的时钟源
01=定时器3是用于CCP2的比较/捕获的时钟源
定时器1是用于CCP1的比较/捕获的时钟源
1x=定时器3是用于CCP模块的比较/捕获的时钟源

T3CKPS1:T3CKPS0 D5 D4 定时器3输入时钟的预分频器值选择位
00=1:1 预分频器值
01=1:2 预分频器值
10=1:4 预分频器值
11=1:8 预分频器值

T1SYNC D2 定时器3的外部时钟输入同步控制位
仅当TMR3CS=1时使用,时钟来自外部源
如果TMR3CS=0,该位不使用
1=不与外部输入时钟同步
0=与外部输入时钟同步

TMR3CS D1 定时器3的时钟源选择位
1=来自T1OSI或者T1CKI的外部时钟源
0=内部时钟($F_{osc}/4$)

TMR3ON D0 定时器3开关控制位
1=使能(启动)定时器3
0=停止定时器3

图9-15 T3CON(定时器3控制)寄存器

						TMR3IF	

TMR3IF 定时器3中断溢出标志位
0=定时器3无溢出
1=定时器3溢出(从FFFF到0000时)

TMR3IF的重要性:在16位模式下,当TMR3H:TMR3L从FFFF变到0000溢出时,该位会变为高电平
PIR寄存器中TMRxIF的位置在以后的产品中可能会不同

377

图9-16 PIR2(外部中断标志寄存器2)包含TMR3IF标志位

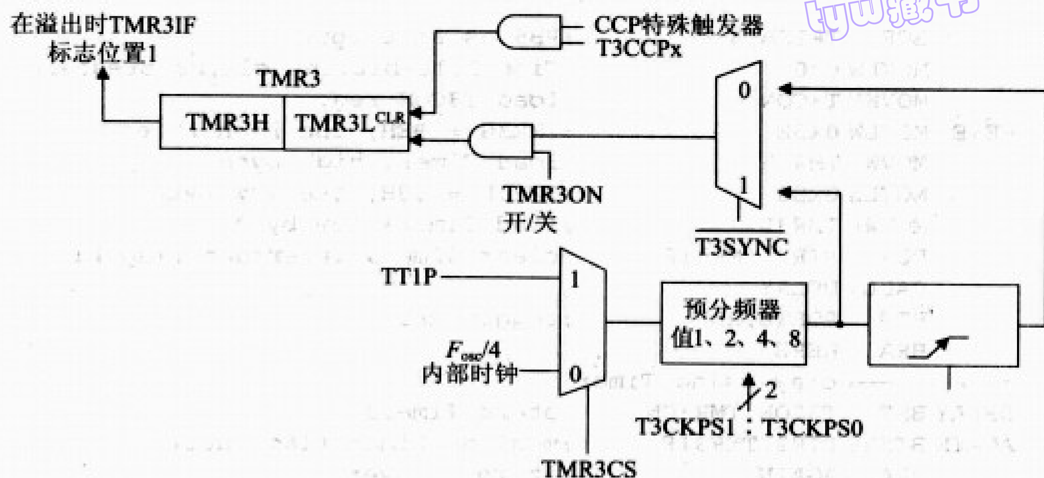


图 9-17 定时器 3 块状图

例 9-42 在 XTAL=10 MHz 时，计算下面程序产生的方波频率。在计算中忽略循环的指令消耗。

```
        BCF    TRISB,5          ;PB5 as an output
        MOVLW  0x0              ;Timer3,16-bit,int clk,no prescale
        MOVWF  T3CON            ;load T3CON reg
HERE    MOVLW  0x76             ;TMR3H = 76H, the high byte
        MOVWF  TMR3H            ;load Timer3 high byte
        MOVLW  0x34             ;TMR3L = 34H, the low byte
        MOVWF  TMR3L            ;load Timer3 low byte
        BCF    PIR2,TMR3IF      ;clear timer interrupt flag bit
        CALL   DELAY
        BTG    PORTB,RB5        ;toggle PB5
        BRA    HERE             ;load TH, TL again
;-----------delay using Timer3
DELAY   BSF    T3CON,TMR3ON     ;start Timer3
AGAIN   BTFSS  PIR2,TMR3IF      ;monitor Timer3 flag until
        BRA    AGAIN            ;it rolls over
        BCF    T3CON,TMR3ON     ;stop Timer3
        RETURN
```

解：

因为 FFFFH−7634H=89CBH+1=89CCH，而 89CCH=35 276 个时钟，35 276×0.4 μs=14.11 ms，频率=1/(14.11 ms×2)=1/28.22 ms=35.434 Hz。这里忽略了循环中的指令消耗。计算过程和例 9-20 一样。

378

例 9-43 假设 XTAL=10 MHz。编制程序，在引脚 PORTB5 上产生频率为 50 Hz 的方波。使用定时器 3，16 位模式，使用允许的最大预分频器值。

解：

因为 FFFFH−9E58H=61A7H+1=61A8H，61A8H=25 000 个时钟，25 000×0.4 μs=1 ms，频率=1/2(1 ms)=50 Hz。计算中忽略了循环中的指令消耗。

```
        BCF   TRISB,5            ;PB5 as an output
        MOVLW 0x0                ;Timer3,16-bit,int clk,no prescale
        MOVWF T3CON              ;load T3CON reg.
HERE    MOVLW 0x9E               ;TMR3H = 9EH, the high byte
        MOVWF TMR3H              ;load Timer3 high byte
        MOVLW 0x58               ;TMR3L = 58H, the low byte
        MOVWF TMR3L              ;load Timer3 low byte
        BCF   PIR2,TMR3IF        ;clear Timer3 interrupt flag bit
        CALL  DELAY
        BTG   PORTB,RB5          ;toggle PB5
        BRA   HERE               ;
;----------delay using Timer3
DELAY   BSF   T3CON,TMR3ON       ;start Timer3
AGAIN   BTFSS PIR2,TMR3IF        ;monitor timer flag until
        BRA   AGAIN              ;it rolls over
        BCF   T3CON,TMR3ON       ;stop timer
        RETURN
```

T=1/50 Hz=2 ms

高低电平的持续时间为 1/2×2 ms=1 ms

需要计数的时钟数 1 ms/0.4 μs=25 000

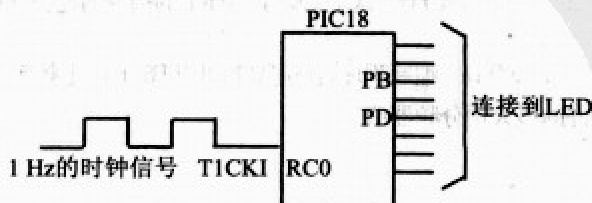
379

65 536−25 000=40 536=9E58H

例 9-44 假设将一个1 Hz的脉冲连接到定时器3的输入(引脚RC0)。编制程序,在PORTB和PORTD上显示TMR3H和TMR3L的值。初值设为0。不使用预分频器。

解:

```
        BSF   TRISC,0            ;PORTC.0 as input T1CLKI
        CLRF  TRISB              ;PORTB as an output
        CLRF  TRISD              ;PORTD as an output
        MOVLW 0x02               ;Timer3,16-bit,ext clk,no prescale
        MOVWF T3CON              ;load T0CON reg
HERE    MOVLW 0x0                ;TMR3H = 0, the low byte
        MOVWF TMR3H              ;load Timer3 high byte
        MOVLW 0x0                ;TMR1L = 0, the low byte
        MOVWF TMR3L              ;load Timer3 low byte
        BCF   PIR2,TMR3IF        ;clear timer interrupt flag bit
        BSF   T3CON,TMR3ON       ;start timer
AGAIN   MOVFF TMR3L,PORTB        ;display low byte count
        MOVFF TMR3H,PORTD        ;display high byte count
        BTFSS PIR2,TMR3IF        ;monitor Timer3 flag until
        BRA   AGAIN              ;it rolls over
        BCF   T3CON,TMR3ON       ;stop Timer3
        GOTO  HERE
```



380

例 9-45 编制 C18 程序，在引脚 PORTB.1 上产生 2500 Hz 的频率。使用定时器 3 来产生时延。

解：

```
#include <p18f4580.h>
void T3Delay(void);
#define mybit PORTBbits.RB1

void main(void)
  {
    TRISBbits.TRISB1=0;          //PB1 as an output
    T3CON=0x00;             //Timer3, 16-bit mode, no prescaler
    while(1)
      {
        mybit=~mybit;            //toggle PB.1
        T3Delay();
      }
  }

void T3Delay()
  {
    TMR3H=0xFE;                  //load TH3
    TMR3L=0x0C;                  //load TL3
    T3CONbits.TMR3ON=1;          //turn on T3
    while(PIR2bits.TMR3IF==0);   //wait for TF3 to roll over
    T3CONbits.TMR3ON=0;          //turn off T3
    PIR2bits.TMR3IF=0;           //clear TF3
  }
```

1/2500 Hz=400 μs

400 μs/2=200 μs

200 μs/0.4 μs=500

65 536−500=65 036=FE0CH

381

例 9-46 假设将一个 1 Hz 的外部时钟输入到引脚 T3(RC0)。编制 C18 程序，让计数器 3 用作计数器。它将记录脉冲数，并在 PORTD 和 PORTB 上分别显示 TMR3H 和 TMR3L 寄存器的内容。

解：

```
#include <p18f4580.h>

void main(void)
  {
      TRISCbits.TRISC0=1;          //make RC0 an input for T1CKI
      TRISB = 0;                   //make PORTB an output
      TRISD = 0;                   //make PORTD an output
      T3CON=0x02;       //Timer1, 16-bit mode, no prescaler
      TMR3H=0;                     //set count to 0
      TMR3L=0;                     //set count to 0
      while(1)                     //repeat forever
       {
        do
         {
           T3CONbits.TMR3ON=1;     //turn on T3
```

```
            PORTB=TMR3L;              //place value on pins
            PORTD=TMR3H;              //
          }
      while(PIR2bits.TMR3IF==0);      //wait for TF3 to roll over
      T3CONbits.TMR3ON=0;             //turn off T3
      PIR2bits.TMR3IF=0;              //clear TF3
    }
}
```

PIC18

将PB和PD连接到LED

1 Hz的时钟信号 T0CKI RC0

9.4.3 复习题

1. 由什么向定时器2提供时钟脉冲?
2. 说明在 T2CON=0x00 时所做的配置选择。
3. 判断对错:定时器2不能用作计数器。
4. 对于定时器3,当它从_______变为_______时计数器复位为零。
5. 如果置 PR2=200,请指出 TMR2IF 何时变为高电平。
6. TMR2IF 和 TMR3IF 都是_______寄存器的一部分。

小结

根据型号的不同,PIC18 可以有 4 个或者更多的定时器/计数器。当用作定时器时,它们可以产生时延。当用作计数器时,它们可以用作事件计数器。本章介绍了定时器/计数器在各种模式下的编程。

通常,定时器可以当作两个寄存器(TMRLx 和 TMRHx)来访问。它们可以用作 16 位定时器或者 8 位定时器。

每个定时器都有自己的 TCON(定时器控制)寄存器,这可以用来选择各种操作模式。这些模式包括预分频器和定时器/计数器选项。当定时器用作定时器时,PIC18 的晶振就是频率源($F_{osc}/4$);而当它用作计数器时,PIC18 的外部脉冲让 TMRxH、TMRxL寄存器增加。

习题

1. PIC18F458 有多少个定时器?
2. PIC18 的定时器 0 是_______位的,可以当作_______和_______访问。
3. PIC18 的定时器 1 是_______位的,可以当作_______和_______访问。

4. 定时器 0 支持的最高预分频器值是________。
5. 定时器 1 支持的最高预分频器值是________。
6. T0CON 寄存器是一个________位寄存器。
7. T0CON 寄存器的工作是什么?
8. 判断对错:T1CON 是位可寻址的寄存器。
9. 计算 T1CON 的值:16 位模式,不使用预分频器,定时器晶振关闭,时钟来自 PIC18 的晶振。
10. 在 PIC18 使用下列的晶振时,计算定时器的频率和周期。
    (a) XTAL=10 MHz (b) XTAL=20 MHz
    (c) XTAL=24 MHz (d) XTAL=30 MHz
11. 对于下面的定时器,请指出哪个寄存器包括 TMRxIF(定时器中断标志) 位。
    (a) 定时器 0 (b) 定时器 1

12. 对于下面的工作模式,请指出定时器的复零值(十六进制和十进制):
    (a) 16 位 (b) 8 位
13. 对于下面的工作模式,请指出 TMR0IF 标志位何时变为高电平。
    (a) 16 位 (b) 8 位
14. 判断对错:定时器 0 和定时器 1 都有自己的定时器中断标志位。
15. 判断对错:定时器 0 和定时器 1 都有自己的定时器启动标志位。
16. 假设 XTAL=10 MHz。计算用于产生 2 ms 时延的 TMR0H 和 TMR0L 的值。使用 16 位模式,不使用预分频器。
17. 假设 XTAL=10 MHz。计算用于产生 5 ms 时延的 TMR0H 和 TMR0L 的值。使用 16 位模式,使用最大的预分频器值。
18. 假设 XTAL=10 MHz。计算用于产生 2.5 ms 时延的 TMR0H 和 TMR0L 的值。使用最大的预分频器值。
19. 假设 XTAL=10 MHz。计算用于产生 0.2 ms 时延的 TMR0H 和 TMR0L 的值。使用 16 位模式,不使用预分频器。
20. 假设 XTAL=20 MHz。计算用于产生 2 ms 时延的 TMR0H 和 TMR0L 的值。使用 16 位模式,使用最大的预分频器值。
21. 假设 XTAL=10 MHz。若要在引脚 RB7 上产生一个方波,请确定用定时器 0 的 16 位模式能得到的方波的最低频率。
22. 假设 XTAL=10 MHz。若要在引脚 RB2 上产生一个方波,请确定用定时器 0 的 16 位模式能得到的方波的最高频率。
23. 使用 8 位模式,重做第 21 题和第 22 题。
24. 在 8 位模式下,假设 TMR0L=F1H,指出定时器 0 在 TMR0IF 变为高电平之前历经的状态。总共有多少个状态?
25. 编制程序,让定时器 0 产生 1 kHz 方波。假设 XTAL=10 MHz。
26. 编制程序,让定时器 1 产生 3 kHz 方波。假设 XTAL=10 MHz。使用最大的预分频器值。
27. 指出定时器 0 和定时器 1 的区别。
28. 确定下面指令送入到 WREG 中的值(十六进制):
    (a) MOVLW - D'12' (b) MOVLW - D'22'
    (c) MOVLW - D'34' (d) MOVLW - D'92'
    (e) MOVLW - D'120' (f) MOVLW - D'104'

29. 将定时器用作计数器，必须把T0CON寄存器的________位设置为________（低电平，高电平）。
30. 定时器0和定时器1都可以用作事件计数器吗？
31. 对于计数器0，哪个引脚用作时钟输入？
32. 对于计数器1，哪个引脚用作时钟输入？
33. 将定时器1编程为事件计数器。使用16位模式，连续地显示PORTB和PORTD的二进制计数值。计数初值设为20 000。
34. 将定时器0编程为事件计数器。使用8位模式，连续地显示PORTB的二进制计数值。计数初值设为20。

384

35. T1CON寄存器是一个________位的寄存器。
36. 判断对错：T1CON寄存器不是位可寻址的寄存器。
37. 使用C语言对定时器0编程，来产生1 kHz方波。假设XTAL=10 MHz。
38. 使用C语言对定时器1编程，来产生1 kHz方波。假设XTAL=10 MHz。
39. 使用C语言对定时器0编程，来产生3 kHz方波。假设XTAL=10 MHz。
40. 使用C语言对定时器1编程，来产生3 kHz方波。假设XTAL=10 MHz。
41. 使用C语言对定时器1编程，将定时器1用作事件计数器，使用16位模式，连续地显示PORTB和PORTD的二进制计数值。计数初值设为20 000。
42. 使用C语言对定时器0编程，将定时器0用作事件计数器，使用8位模式，连续地显示PORTB的二进制计数值。计数初值设为20。
43. 指出下面的定时器所支持的预分频器的最大值：
    (a) 定时器2　　(b) 定时器3
44. 指出定时器3的复零值（十六进制和十进制）。
45. 对于下面的定时器，请指出定时器标志位在什么时候变为高电平。
    (a) 定时器2　　(b) 定时器3
46. 判断对错：定时器2的PR2寄存器是一个8位寄存器。
47. 判断对错：定时器2和定时器3都是16位的定时器。
48. 假设XTAL=10 MHz。计算用于产生2 ms时延的TMR3H和TMR3L的值。不使用预分频器。
49. 假设XTAL=10 MHz。计算用于产生5 ms时延的TMR3H和TMR3L的值。使用最大的预分频器值。
50. 将定时器3编程为事件计数器。使用16位模式，连续地显示PORTB和PORTD的二进制计数值。计数初值设为20 000。
51. 使用汇编语言对定时器2编程，让它从0计数到200时翻转引脚RB3。假设XTAL=10 MHz。
52. 使用C语言对定时器2编程，产生3 kHz的方波。假设XTAL=10 MHz。
53. 使用C语言对定时器2编程，让它从0计数到200时翻转引脚RB3。假设XTAL=10 MHz。
54. 使用C语言对定时器3编程，产生1 kHz的方波。假设XTAL=10 MHz。

385

复习题答案

9.1节

1. 4　　2. 错误。　　3. 正确。　　4. 正确。
5. 0000 1000代表使用16位模式，使用预分频器，使用XTAL作为频率信号。
6. FFFFH，0000　　7. FFH，00

8. −200 的十六进制数为 38H；因此，WREG=38H。

9. 2 ms/0.4 μs=5000，65 536−5000=60 536=EC78H，TMR0H=ECH，而 TMR0L=78H。

10. 100 μs /0.4 μs=250，256−250=06；因此，TMR0L=06H。

9.2 节

1. 连接到 PIC18 的晶振。 2. 定时器的时钟源来自引脚 RA4(PORTA4)。

3. 是的。 4. 必须把引脚设为输入，以让外部源时钟输入。 5. 是的。

9.4 节

1. 连接到 PIC18 的晶振($F_{osc}/4$) 2. 预分频器值为 1，后分频器值为 1，停止定时器 2。

3. 正确。 4. FFFFH，0

5. TMR2 计数直到等于 PR2 的值。此时，TMR2IF 变为高电平。 6. 分别是 PIR1 和 PIR2。

386

第 10 章 PIC18 串行端口的汇编编程和 C 编程

学习目标：

- ☐ 串行通信和并行通信
- ☐ 串行通信优于并行通信之处
- ☐ 串行通信的协议
- ☐ 同步通信和异步通信
- ☐ 半双工传输和全双工传输
- ☐ 数据帧的过程
- ☐ 数据传输速度和波特率
- ☐ RS232 标准
- ☐ MAX232 和 MAX233 芯片的应用
- ☐ PIC18 与 RS232 连接器的接口
- ☐ 关于 PIC18 的波特率
- ☐ PIC18 的串行通信特征
- ☐ PIC18 用于串行通信的主要寄存器
- ☐ 使用汇编和 C 语言对 PIC18 串行端口进行编程

计算机传输数据有两种方式：并行和串行。在并行数据传输中，通常使用 8 根或更多的导线来将数据传输到一个只有几英尺远的设备上。并行传输设备包括打印机和硬盘，它们使用的是有多根线的电缆。虽然并行导线可以在短时间内传输大量的数据，但是传输的距离不能太远。为了能将数据传输到一个几米开外的设备，需要使用串行方法。相比较而言，在串行通信中，每一次只传输一位数据；而并行通信一次可以传输一个字节或者更多的数据。本章将讨论 PIC18 的串行通信。由于 PIC18 有内置的串行通信功能，因此使用很少的导线就能实现数据的快速传输。

在本章中，首先讨论串行通信的基础知识。10.2 节将介绍通过 MAX232 线驱动器来实现 PIC18 和 RS232 连接器的连接方法。10.3 节将会讨论 PIC18 的串行端口编程。10.4 节将介绍如何使用 C18 编译器对 PIC18 串行端口进行 C 语言编程。

10.1 串行通信基础

当微处理器与外部世界进行通信时，它提供的是字节大小的数据块。在有些情况下（如使用打印机时），只是简单地读取来自 8 位数据总线的信息，然后再传送给打印机的 8 位数据

总线。这种数据的传输方式只适用于导线不是很长的情况,因为长导线会衰减甚至扭曲信号。而且,8 位数据线是很昂贵的。鉴于这些原因,串行通信常用于相距几十米甚至几百万米的两个系统之间的数据传输。图 10-1 分别画出了串行和并行数据传输方式。

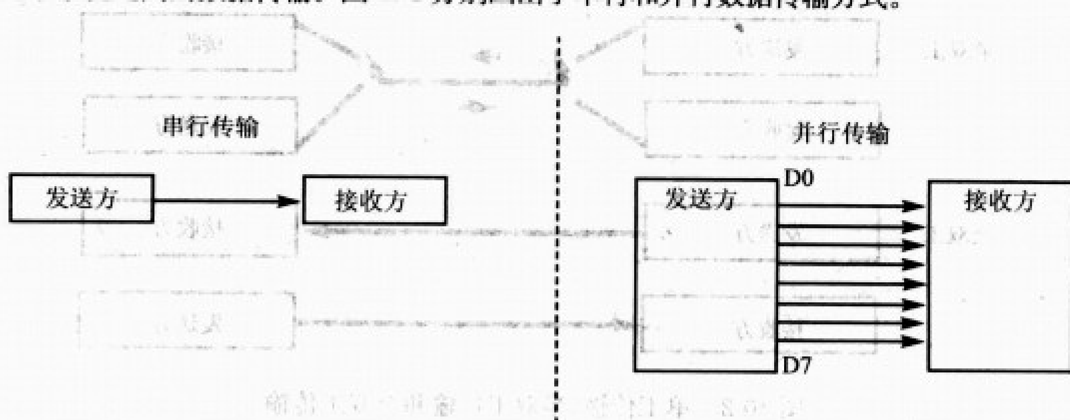


图 10-1 串行数据传输和并行数据传输

实际上,使用串行通信的单数据线代替并行通信的 8 位数据线,不仅使得串行传输更经济,而且还能让位于两个城市的计算机通过电话线通信。

要使得串行通信能工作,必须首先将字节数据通过"并入串出"移位寄存器转换成串行位数据;然后信息才能在单数据线路上传送。这也说明,在接收端必须有一个"串入并出"移位寄存器来接收串行数据,并将它们组合成字节数据。当然,如果数据是通过电话线传输的,必须将它们从 0 和 1 转化成语音信号,也就是正弦信号。这个转换过程是由一个叫作调制/解调器的外部设备完成的。 388

当传输距离很短时,数字信号可以在简单线路上传输而不需要调制。IBM PC 键盘就是以这种方式传输数据到主板的。对于长距离的数据传输(如电话),串行数据通信就需要用到调制/解调器来调制(把 0 和 1 转化成语音信号)和解调(把语音信号转化成 0 和 1)。

串行数据通信有两种方法:异步通信和同步通信。同步方法是指一次传输一组数据(字符),而异步方法是指一次只传输一个字节。这两种方法都可以采用软件编程来实现,但是所编写的程序将会很繁琐。为此,许多公司生产出特殊的 IC 芯片,用于串行数据通信。这些芯片通常被称作 UART(通用异步收发机)或者是 USART(通用同异步收发机)。PIC18 芯片带有内置的 USART,这将在 10.3 节中详细地讨论。

10.1.1 半双工和全双工传输

在数据传输过程中,如果双方都可以接收和发送数据,这就被称作*双工传输*。这是相对于*单工传输*(如打印机)而言,在单工传输中计算机只能发送数据。双工传输又可以分为半双工和全双工,这取决于数据的传输是否同时进行。如果数据每次只能单向传输,就叫作*半双工*。如果数据每次都可以双向传输,就叫作*全双工*。当然,为了能同时发送和接收数据,全双工需要两路数据导线(接地信号线除外),一路用于发送而另一路用于接收。如图 10-2 所示。 389

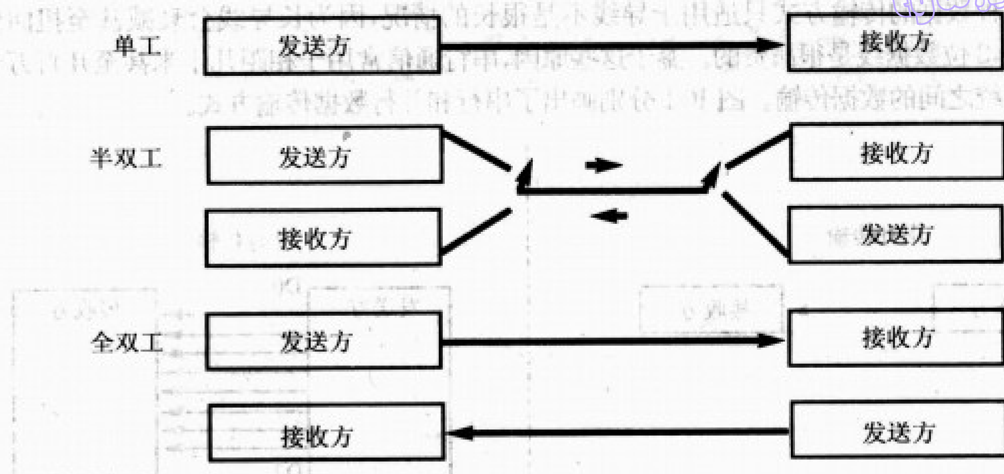


图 10-2　单工传输、半双工传输和全双工传输

10.1.2　异步串行通信和数据帧

当串行传输的数据到达线路末端的接收方时，它是以0和1的形式表示的。这样的数据是很难被识别的，除非发送者和接收者都遵守一定的规则(即协议)如怎样打包数据，使用多少位表示一个字符，以及数据何时开始和结束。

10.1.3　起始位和结束位

异步串行数据通信广泛地用于面向字符的数据传输，而同步传输使用的是面向数据块的传输。在异步传输中，每个字符都放置在起始位和结束位之间，这称为帧。在异步通信的数据帧中，数据(如ASCII字符)是放置在一个起始位和一个结束位之间的。起始位都是一位的，而结束位可能是一位或者两位的。起始位总是0(低电平)，结束位是1(高电平)。例如，如图10-3所示，ASCII字符A(8位二进制0100 0001)就被封装在起始位和单一的结束位之间。注意：先发送LSB。

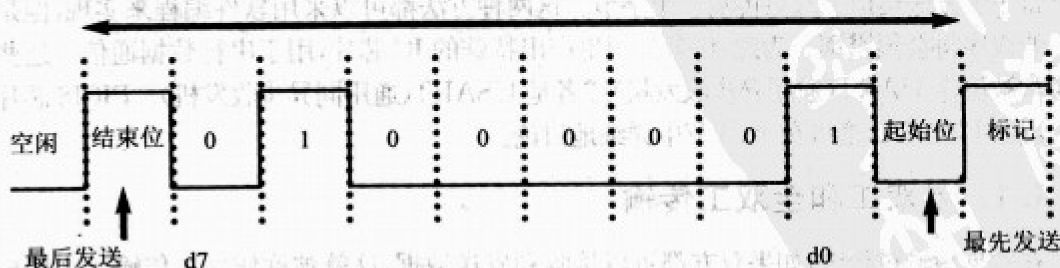


图 10-3　ASCII A(41H)的帧

注意，在图10-3中，当无发送数据时，信号为1(高电平)，被称为标记。而0(低电平)则被称为空闲。要记住的是，首先传输起始位，然后传输D0(LSB)，接着传输其余位直到MSB(D7)，最后是传输结束位，标志着字符A的结束。

在异步串行通信中，外部芯片和调制/解调器都支持7位或者8位长的数据编程，这不包

括结束位的数量(1 或者 2)。在以前的系统中,ASCII 字符是 7 位的,而近年来,因为扩展 ASCII 字符的出现,8 位数据成为主流。在过去的系统中,因为接收设备的物理速度较慢,所以使用 2 位的结束位让设备在传送下一字节之前有足够的时间组织数据。而对于现在的 PC,都是用标准的 1 位结束位。假设在传输一个 ASCII 字符文本时使用 1 位的结束位,那么每个字符就是 10 位大小:包括 8 位的 ASCII 代码、1 位的起始位和 1 位结束位。因此,每个 8 位的字符有 2 个附加位,也就是 25%的额外开销。

390

在一些系统里,数据帧也会包含字符字节的奇偶校验位,以保持数据完整性。也就是说,每个字符(到底是 7 位或 8 位,由系统决定)除了起始位和结束位之外,还有一个奇偶校验位。奇偶校验位可以是奇数或者偶数。若是奇校验,则包括校验位在内,数据帧中 1 的位数应该是单数。相似地,若是偶校验,则包括校验位在内的 1 位数应是偶数。例如,ASCII 字符 A,其二进制码是 0100 0001,0 作为它的偶校验位。UART 芯片可以被编程为奇校验、偶校验或者是无校验。

10.1.4 数据传输率

串行通信的数据传输速率通常用 bit/s(位每秒)来表示。另一个常用的 bit/s 术语是伯德率。不过伯德率和 bit/s 率不是绝对相等的。这是因为伯德率是调制/解调器术语,用来定义每秒的信号转换率。在调制/解调器中,有时候每次信号转换会传输几位的数据。就已介绍过的导线而言,伯德率和 bit/s 是相等的,因此本书中使用的术语 bit/s 和伯德率是可以互换的。

给定的计算机系统的数据传输速率由连接到系统的通信端口决定。例如,早期的 IBM PC/XT 可以以 100 bit/s~9600 bit/s 的速率传输数据。近年来,基于 Pentium 的 PC 的数据传输速率可以高达 56 Kbit/s。注意,在异步串行数据通信中,波特率的上限通常是 100 000 bit/s。

10.1.5 RS232 标准

为了使得不同厂家生产的数据通信设备能彼此兼容,电子工业联盟(EIA)在 1960 年制定了一个接口标准 RS232。在 1963 年,这个标准又被修订成 RS232A;后来又分别在 1965 年和 1969 年被修订成 RS232B 和 RS232C。在本书中,我们笼统称它们为 RS232。现在,RS232 是最通用的串行 I/O 接口标准。这个标准可以应用在 PC 和很多类设备上。由于该标准是在 TTL 逻辑器件发明之前制定的,所以它的输入和输出电压并不兼容于 TTL。在 RS232 里,1 代表−3 V~−25 V,0 表示+3 V~+25 V,而−3 V~+3 V 是未定义的。因此,要把 RS232 器件连接到微控制器系统,必须进行电压转换,如使用 MAX232 把 TTL 逻辑电平转换到 RS232 电平,反之亦然。MAX232 IC 芯片通常用作线驱动。关于 RS232 和 MAX232 的连接将会在 10.2 节中讨论。

10.1.6 RS232 引脚

表 10-1 给出了 RS232 线缆的引脚和标号,这通常是指 DB-25 连接器。在标号中,DB-25P 表示插头连接器(公端),DB-25S 表示插座连接器(母端)。如图 10-4 所示。

391

由于并非所有的引脚都用于 PC 电缆,所以 IBM 引入了 DB-9 的串行 I/O 标准,也就是只

使用 9 个引脚，如表 10-2 所示。DB-9 的引脚如图 10-5 所示。

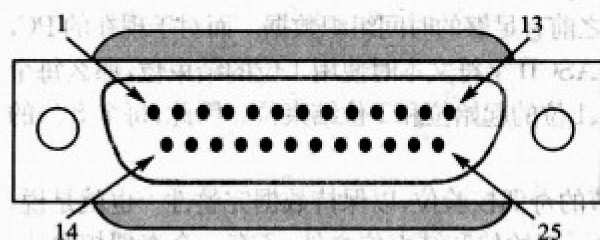


图 10-4 RS232 连接器 DB-25

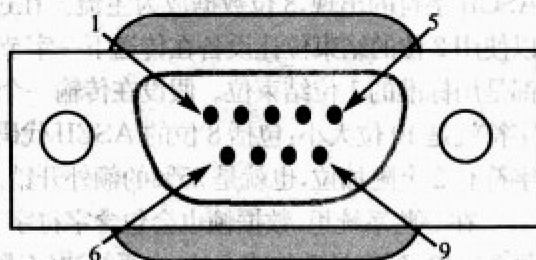


图 10-5 DB-9 9 引脚连接器

表 10-1 RS232 引脚(DB-25)

引 脚	描 述	引 脚	描 述
1	接地保护	14	辅助数据传输
2	数据传输(TxD)	15	信号传送时序
3	数据接收(RxD)	16	辅助数据接收
4	发送请求($\overline{\text{RTS}}$)	17	信号接收时序
5	发送清除($\overline{\text{CTS}}$)	18	未分配
6	数据设备就绪($\overline{\text{DSR}}$)	19	辅助发送请求
7	信号接地(GND)	20	数据终端就绪($\overline{\text{DTR}}$)
8	数据载波检测($\overline{\text{DCD}}$)	21	信号质量监测
9/10	用于数据检测的保留	22	响铃指示
11	未分配	23	数据信号速率选择
12	辅助数据载波检测	24	信号传送时序
13	辅助发送清除	25	未分配

10.1.7 数据通信的分类

目前，学术上将数据通信设备分成 DTE(数据终端设备)和 DCE(数据通信设备)。DTE 指发送和接收数据的终端和计算机，而 DCE 指的是通信设备，如用于传输数据的调制/解调器。注意，表 10-1 和表 10-2 都是从 DTE 角度来定义 RS232 引脚功能的。

PC 和微控制器之间最简单的连接最少需要 3 个引脚，即 TX、RX 和接地，如图10-6所示。注意，在该图中 RX 和 TX 可以互换。

表 10-2 IBM PC DB-9 信号

引 脚	描 述	引 脚	描 述
1	数据载波检测($\overline{\text{DCD}}$)	6	数据发送准备就绪($\overline{\text{DSR}}$)
2	接收数据(RxD)	7	请求发送($\overline{\text{RTS}}$)
3	发送数据(TxD)	8	清零发送($\overline{\text{CTS}}$)
4	数据接收端准备就绪(DTR)	9	响铃指示(RI)
5	信号接地(GND)		

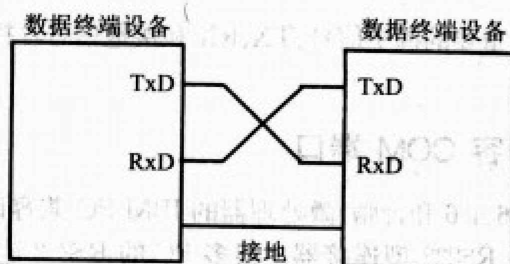


图 10-6 不使用调制/解调器的连接

10.1.8 检查 RS232 的握手信号

为了确保两个设备之间高速、可靠的数据传输，数据传输必须协调地进行。以打印机为例，因为接收设备可能没有给串行通信数据提供存储空间，所以必须要能通知发送方停止发送数据。RS232 连接器的很多引脚都是用于握手信号的。下文关于它们的描述仅供参考，对于不被 PIC18 UART 芯片支持的那些引脚是可以忽略的。

392

(1) DTR(数据终端就绪)。当终端(或者是 PC 的 COM 端口)被打开时，在经过自身检测后，它向外发送 DTR 信号来声明自己已经准备就绪。如果 COM 端口出现问题，这个信号将不会出现。它是一个低电平有效的信号，可以用来让调制/解调器知道计算机正在活动。这是 DTE(PC COM 端口)的输出引脚和调制/解调器的输入引脚。

(2) DSR(数据设备就绪)。当 DCE(调制/解调器)启动时，在经过自身检测后，它使用 DSR 来声明自己已经准备就绪。因此，它是调制/解调器(DCE)的输出和 PC(DTE)的输入。该信号是低电平有效。如果因为任何原因导致调制/解调器不能连通到电话，那么这个信号将保持无效状态，以向 PC(或者是终端)表明它不能接收或发送数据。

(3) RTS(发送请求)。当 DTE 设备(如 PC)有一个字节数据要发送时，它使用 RTS 来告诉调制/解调器它有一个字节的数据需要传输。RTS 是一个来自 DTE 的输出和调制/解调器的输入，是低电平有效的。

(4) CTS(发送清除)。作为对 RTS 信号的响应，在调制/解调器有空间来存储将要接收的数据时，它就发出 CTS 信号到 DTE(PC)，以声明它现在可以接收数据。DTE 用该输入信号启动传输信号。

(5) DCD(数据载波检测)。调制/解调器使用 DCD 信号来通知 DTE(PC)检测到一个有效载波，而且它已和其他调制/解调器建立起连接。因此，DCD 是调制/解调器的输出信号和 PC(DTE)的输入信号。

(6) RI(响铃指示)。调制/解调器(DCE)的输出和 PC(DTE)的输入表明电话在响铃。RI 信号的续和断与响铃声同步。在 6 个握手信号中，这是最不常用的，因为调制/解调器注重电话信号的应答。然而，如果在一个系统里 PC 负责控制电话的应答，那么就会使用这个信号。

393

根据以上的描述，PC 和调制/解调器之间的通信可以总结为：DTR 和 DSR 分别是 PC 和调制/解调器用来声明自己准备就绪的，而 RTS 和 CTS 是用来控制数据流的。当 PC 要发送数据时，它会发送 RTS 信号。作为应答，如果调制/解调器准备就绪(有空间)接收数据，则它将回传 CTS 信号。如果缺少空间，调制/解调器就不会发出 CTS 信号，PC 将取消 DTR，并重试。RTS 和 CTS 也可用作硬件控制流信号。

这里描述了RS232最重要的握手信号、TX、RX和接地信号。接地信号也称为SG(信号接地)。

10.1.9 IBM PC/兼容COM端口

基于x86(8086、286、386、486和奔腾)微处理器的IBM PC/兼容计算机都有两个COM端口。两个COM端口都是RS232型连接器。很多PC的RS232连接器都使用DB-25或者DB-9。这两个COM端口常被记作COM1和COM2。近年来,其中一个端口为USB端口所替代,而COM1只用于串行端口。读者可以将PIC18串行端口接到PC的COM1端口,用于串行通信实验。如果没有COM端口,可以使用COM-USB转换器。

具备了串行通信的背景知识,接下来将介绍PIC18的串行通信。在下一节中将讨论PIC18和RS232连接器的物理连接,在10.3节将会讨论PIC18串行通信端口的编程。

10.1.10 复习题

1. 使用并行线路传输数据会________(更快,更慢),不过成本会________(更高,更低)。
2. 判断对错:传输数据到打印机是双工的。
3. 判断对错:对于全双工,必须有两条数据线,一条用于发送,一条用于接收。
4. 起始位和结束位是用在________(同步,异步)传输中的。
5. 假设要传输ASCII的字母"E"(二进制的0100 0101),带1位结束位,无奇偶校验位,请写出串行传输时的位序列。
6. 在第5题中,请计算数据帧的额外开销。
7. 如果在第5题中要传输10 000个字符,速率是9600 bit/s,请计算所需要的时间。其中有多少百分比的时间是用于额外开销的?
8. 判断对错:RS232和TTL是不兼容的。
9. 在RS232协议中,二进制0表示的电压是多少?
10. 判断对错:PIC18带有片内UART。
11. 在x86 PC上,通常有________个COM端口连接器。
12. PC的COM端口被DOS和Windows指定为________和________。

394

10.2 PIC18连接到RS232

本节将会具体介绍PIC18与RS232连接器的物理连接。如10.1节所介绍的,RS232标准对于TTL并不兼容;因此,需要使用线驱动器(如MAX232芯片)来把RS232电压转换成TTL电平,反之亦然。通过MAX232芯片来实现PIC18和RS232连接器的连接是本节的主要内容。

10.2.1 PIC18的RX和TX引脚

PIC18有两个专门用于连续发送和接收数据的引脚,它们分别是TX和RX,位于40脚封装的PORTC组(RC6和RC7)。PIC18的引脚25(RC7)是分配给TX的,而引脚26(RC6)是分配给RX的。这两个引脚是TTL兼容的;因此它们需要一个线驱动器来使它们和RS232兼

容。MAX232 芯片就是实现这种功能的一种线驱动器。下面将对它作以介绍。

10.2.2 MAX232

因为 RS232 与现在的微处理器和微控制器不兼容，所以需要使用线驱动器（电压转换器）来将 RS232 的信号转换成 TTL 电平，以适用于 PIC18 的 TX 和 RX 引脚。Maxim 公司(www.maxim-ic.com)的 MAX232 就是实现这种功能的一种转换器。MAX232 将 RS232 的电压转换成 TTL 电平，反之亦然。使用 MAX232 芯片的一个好处在于它使用的是＋5V 电源，这和 PIC18 的电源电压相同。换句话说，一个＋5 V 的电压就可以同时为 PIC18 和 MAX232 提供电源，而不需要使用像在很多旧系统中常见到的双电源。

MAX232 有两套线驱动器，分别用于发送和接收数据，如图 10-7 所示。用于 TX 的线驱动器叫作 T1 和 T2，而用于 RX 的线驱动器是 R1 和 R2。在很多应用中，只会用到其中的一套线驱动器。例如，T1 和 R1 一起用于 PIC18 的 TX 和 RX，而另一套则不会使用。注意，MAX232 的 T1 线驱动器指定引脚 11 和 14 分别为 T1in 和 T1out。T1in 引脚在 TTL 侧，连接到微控制器的 TX，而 T1out 引脚在 RS232 侧，连接到 RS232 DB 连接器的 RX 引脚。R1 线驱动器指定引脚 13 和 12 分别为 R1in 和 R1out。R1in(引脚 13)在 RS232 侧，连接到 RS232DB 的 TX 引脚，而 R1out(引脚 12)在 TTL 侧，连接到微控制器的 RX 引脚。如图 10-7 所示。注意，在不使用调制/解调器连接时，两个引脚分别用于 RX 和 TX。

395

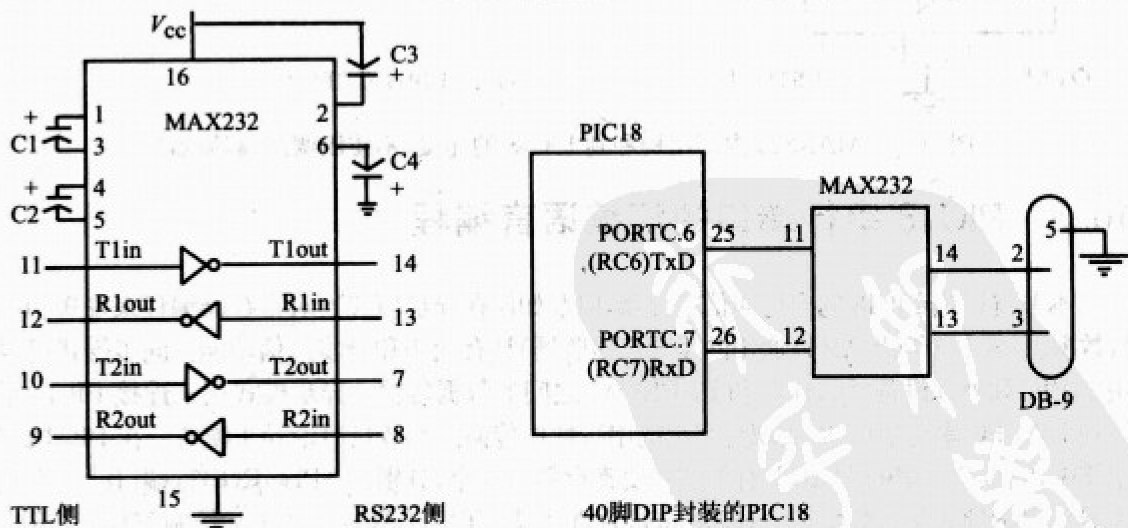


图 10-7　MAX232 内部结构和到 PIC18 的连接（不使用调制/解调器）

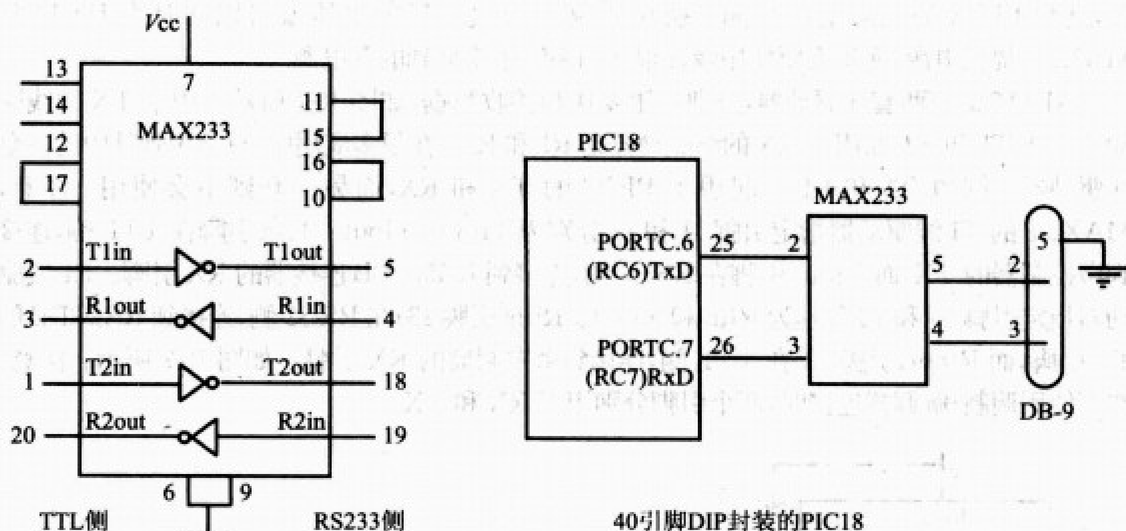
MAX232 需要用到 4 个电容，电容值范围从 1 μF 到 22 μF。最常用的电容值是 22 μF。

10.2.3 MAX233

为了节省线路板空间，有些设计者会选用 Maxim 的 MAX233 芯片。MAX233 可以完成和 MAX232 一样的功能，只不过不再使用电容而已。不过，MAX233 芯片比 MAX232 要贵得多。注意，MAX233 和 MAX232 的引脚是不兼容的。不能把电路板上的 MAX232 芯片取出而用 MAX233 来代替。如图 10-8 所示，MAX233 没有使用电容。

10.2.4 复习题

1. 判断对错:PC COM 端口连接器是 RS232 类型的。
2. PIC18 的哪些引脚是用于串行通信的,它们的功能分别是什么?
3. 线驱动器(如 MAX232)的作用是什么?
4. MAX232 可以支持 TX 的________线和 RX 的________线。
5. MAX233 相比 MAX232 的优势在哪里?



396

图 10-8 MAX233 内部结构和到 PIC18 的连接(不使用调制/解调器)

10.3 PIC18 串行端口的汇编语言编程

本节将讨论 PIC18 的串行通信寄存器,以及如何在异步模式下对寄存器编程来发送和接收数据。PIC18 的 USART(通用同步异步接收器)具有同步和异步通信功能。同步模式可以用于 PIC 和外部设备(如 ADC 和 EEPROM)之间的数据传输。异步模式用来连接 PIC18 系统到 IBM PC 串行端口,以实现全双工的串行数据传输。本节只讨论异步模式。在 PIC 微控制器中,同本章介绍的 UART 有关的主要寄存器有 6 个,分别是 SPBGR(串行端口波特率发生器)、TXREG(发送寄存器)、RCREG(接收寄存器)、TXSTA(发送状态和控制寄存器)、RCSTA(接收状态和控制寄存器)和 PIR1(外部中断请求寄存器 1)。下面将逐个地介绍它们,以及它们在全双工串行通信中的应用。

10.3.1 PIC18 的 SPBRG 寄存器和波特率

因为 IBM PC/兼容计算机经常用于和 PIC18 系统进行通信,所以本节将会重点介绍 PIC18 和 PC 的 COM 端口之间的串行通信。表 10-3 中列出了一些 PC HyperTerminal 支持的波特率。读者可以在微软 Windows 的 HyperTerminal 程序中选择通信设置来验证这些波特率。PIC18 支持以不同的波特率发送和接收数据,这些工作是由 8 位寄存器 SPBRG 完成

的。对于给定的晶振频率，SPBGR 的值决定了串行通信的波特率。SPBGR 的内容和 F_{osc}（连接到 OSC1 和 OSC2 引脚的晶振频率）的关系可由下面的方程给出：

$$期望波特率=F_{osc}/(64X+64)=F_{osc}/64(X+1)$$

其中，X 是 SPBGR 的值。假设 F_{osc}=10 MHz，则有：

$$期望波特率=F_{osc}/(64X+1)=10\ MHz/64(X+1)=6250\ Hz/(X+1)$$

为了得到不同波特率对应的 X 值，我们求解下面的方程：

$$X=(15\ 6250/期望波特率)-1$$

表 10-3 PCHyperTerminal 支持的波特率

1 200	19 200
2 400	38 400
4 800	57 600
9 600	115 200

表 10-4 列出了在 F_{osc}=10 MHz 时不同波特率对应的 X 值。在表 10-4 中，另一种用于理解 SPBRG 值的方法是从指令周期的角度来观察它们。正如在前几章中谈到的，PIC18 把晶振频率（F_{osc}）除以 4，可以得到指令周期信号频率。当 XTAL=10 MHz 时，指令周期频率为 2.5 MHz。在设置内部时钟的波特率之前，PIC18 的 UART 电路将指令周期频率再除以 16。因此，2.5 MHz 除以 16 就等于 156 250 Hz，这是在表 10-4 中可以查到的 SPBRG 值。例 10-1 说明了如何验证表 10-4 中的数据。表 10-5 列出了在晶振频率为 4 MHz（F_{osc}=4 MHz）下的 SPBRG 值。

397

例 10-1 假设 F_{osc}=10 MHz。请计算下面波特率所需的 SPBRG 值：

(a) 9 600 (b) 4 800 (c) 2 400 (d) 1 200

解：因为 F_{osc}=10 MHz，所以可得到 10 MHz/4=2.5 MHz 的指令周期频率。在用于 UART 之前，可再除以 16。因此，有 2.5 MHz/16=156 250 Hz 和 X=(156 250 Hz/期望波特率)−1：

(a) (156 250/9600)−1=16.27−1=15.27=15=F(十六进制)，将被赋值给 SPBRG

(b) (156 250/4800)−1=32.55−1=31.55=32=20(十六进制)，将被赋值给 SPBRG

(c) (156 250/2400)−1=65.1−1=64.1=64=40(十六进制)，将被赋值给 SPBRG

(d) (156 250/1200)−1=130.2−1=129.2=129=81(十六进制)，将被赋值给 SPBRG

注意，指令周期除以 16 是在复位期间设置的。在同样的晶振下，可以通过改变默认设置来获得更高的波特率。具体的操作方法是将 TXSTA 寄存器里的 BRGH 置 1。这将在本节的最后给予解释。



表 10-4 不同波特率对应的 SPBRG 值（F_{osc}=10 MHz，BRGH=0）

波特率	SPBRG（十进制）	SPBRG（十六进制）
38 400	3	3
19 200	7	7
9 600	15	F
4 800	32	20

(续)

波特率	SPBRG (十进制)	SPBRG (十六进制)
2 400	64	40
1 200	129	81

398

注意:对于 F_{osc}=10 MHz,有 SPBRG=(156 250/波特率)−1。

表 10-5 不同波特率对应的 SPBRG 值(F_{osc}=4 MHz,BRGH=0)

波特率	SPBRG (十进制)	SPBRG (十六进制)
19 200	2	2
9 600	5	5
4 800	12	0C
2 400	25	19
1 200	51	33

注意:对于 F_{osc}=4 MHz,有 4 MHz/4=1 MHz 的指令周期频率。因此,UART 使用的频率为 1 MHz/16 =62 500 Hz,也即是 SPBRG=(62 500/波特率)−1。

10.3.2 TXREG 寄存器

TXREG 是 PIC18 用于串行通信的另一个 8 位寄存器。要通过 TX 引脚进行串行传输的字节数据必须放置在 TXREG 寄存器中。TXREG 是一个 SFR,其访问方法和其他 PIC18 寄存器一样。在下面的例子中将会了解到如何访问该寄存器。

```
MOVLW 0x41      ;WREG=41H, ASCII for letter 'A'
MOVWF TXREG     ;copy WREG into TXREG

MOVFF PORTB,TXREG ;copy PORTB contents into TXREG
```

当一个字节写入 TXREG 时,它也会被送入 TSR 寄存器(发送移位寄存器)。TSR 将 8 位数据、起始位和结束位封装在一起,然后将得到的 10 位数据通过 TX 引脚串行地发送。注意,程序员是可以访问 TXREG 寄存器的,但是不能访问 TSR,它仅供内部使用。

10.3.3 RCREG 寄存器

类似地,当 RX 引脚接收到数据时,PIC18 会抛弃结束位和起始位,抽取得到的字节数据,然后将其送入 RCREG 寄存器中。执行下面的代码,将把接收到的字节数据送入 PORTB:

```
MOVFF RCREG,PORTB ;copy RXREG to PORTB
```

10.3.4 TXSTA(发送状态和控制寄存器)

TXSTA 是一个 8 位寄存器,用来选择同步/异步模式和数据帧的大小等。图 10-9 描述了 TXSTA 寄存器的各位。本书使用 8 位大小的异步传输模式。BRGH 位用于选择更高的传输速度,其默认值是较低的波特率传输。更高速度的传输将在本章的最后介绍。注意,TXSTA 寄存器的 D6 位用于指定数据帧中每个字符所占的位数。这里使用的是 8 位数据。在有些应用中还会用到 9 位的数据格式,其中第 9 位用作地址。

399

CSRC	TX9	TXEN	SYNC	0	BRGH	TRMT	TX9D

CSRC　D7　时钟源选择(在异步模式下不使用,因此有 D7=0)

TX9　D6　9 位传输使能位
1=选用 9 位传输
0=选用 8 位传输(这里选择该模式,因此有 D6=0)

TXEN　D5　发送使能位
1=启动发送
0=禁止发送
置该位为 on 启动数据传输,置该位为 off 停止数据传输

SYNC　D4　USART 模式选择(这里使用异步模式,因此有 D4=0)
1=同步
0=异步

0　D3

BRGH　D2　高波特率选择位
0=低速(默认)
1=高速
使用该设置,在相同的 F_{osc} 下可以将波特率提高 1 倍。请参阅在本节的最后部分关于该位设置的讨论

TRMT　D1　发送移位寄存器(TSR)状态
1=TSR 空
0=TSR 满

TSR 寄存器的重要性　要串行传输 1B 的数据,需要将它写入到 TXREG 寄存器。TSR(发送移位寄存器)是一个内部寄存器,用来将 TXREG 寄存器的内容封装上起始位和结束位,然后通过 TX 引脚按位串行发送。当最后一位(即结束位)发送完毕时,TRMT 标志位变为高电平,则表明 TSR 空,可以接收下一个字节数据。当 TSR 从 TXREG 寄存器读取数据时,它会对 TRMT 标志位清零,以表明 TSR 满。注意:TSR 是一个"并入串出"的移位寄存器,程序员不能访问它。程序员唯一能做的就是写 TXREG 寄存器。当 TSR 为空时,它从 TXREG 寄存器读取数据,然后立即清空 TXREG 寄存器,因此不会出现两次发送相同数据的情况

TXD9　D0　传送数据的第 9 位(因为使用的是 8 位模式,所以置 D0=0)
在某些应用中,该位可用作一个地址/数据位,或者是奇偶校验位

图 10-9　TXSTA:发送状态和控制寄存器

10.3.5 RCSTA(接收状态和控制寄存器)

RCSTA 寄存器是一个 8 位寄存器,用来设置串行端口的数据接收以及其他的工作。图 10-10 描述了 RCSTA 寄存器的各位。在本节中使用的是 8 位数据帧。

400

SPEN	RX9	SREN	CREN	ADDE	FERR	OERR	RX9D

SPEN D7 串行端口使能位
1=启用串行端口,TX和RX引脚用作串行端口的引脚
0=禁用串行端口
RX9 D6 9位接收使能位
1=选择9位接收
0=选择8位接收(这里使用该选项,因此有D6=0)
SREN D5 单一接收使能位(在异步模式下不使用,因此有D5=0)
CREN D4 连续接收使能位
1=启用连续接收(异步模式)
0=禁止连续接收(异步模式)
ADDEN D3 地址删除使能位(因为使用的是9位数据帧,因此有D3=0)
FERR D2 数据帧出错位
1=数据帧错误
0=数据帧无错误
OERR D1 溢出错误位
1=溢出错误
0=无溢出错误
TXD9 D0 接收数据的第9位(因为使用的是8位模式,所以置D0=0)
在某些应用中,该位可用作一个地址/数据位,或者是奇偶校验位

图10-10 RCSTA:接收状态和控制寄存器

10.3.6 PIR1(外部中断请求寄存器1)

在第9章中,已经看到PIR1的某些位是可以用作定时器标志位的。PIR1有两个位是供UART使用的,它们是TXIF(发送中断标志位)和RCIF(接收中断标志位),如图10-11所示。在向TXREG寄存器写入新的字节数据之前,需要监视(查询)TXIF标志位,以确定上一字节是否全部发送完毕。同样,也需要检查RCIF标志位,以查看上一字节是否完全接收到。在第11章中,将会看到这些标志位同中断的结合应用。下面将介绍TXIF标志位在串行数据传输中是如何工作的。

10.3.7 PIC18串行数据发送编程

对PIC18字节数据的串行发送进行编程,要遵循下面的步骤。

(1)将TXSTA寄存器赋值为20H,表明使用的是异步传输模式、8位数据帧、低波特率、允许发送。

(2)将PORTC(RC6)的TX引脚用作PIC数据的输出。

(3)将SPBRG赋值为表10-4(或者表10-5,前提是F_{osc}=4 MHz)中的某个值,来设定串行数据传输的波特率。

(4) 将 RCSTA 寄存器中的 SPEN 位设为高电平，启动 PIC18 的串行端口。
(5) 将要发送的字符字节数写入到 TXREG 寄存器。
(6) 监视 PIR1 寄存器的 TXIF 位，确保 UART 准备就绪以接收下一字节数据。
(7) 传输下一个字符，返回到步骤(5)。

—	—	RCIF	TXIF	—	—	—	

RCIF　接收中断标志位
　　1=UART 接收到一个字节数据，占用 RCREG 寄存器(接收缓冲)，等待取出
　　当读 RCREG 寄存器时，RCIF 将被清零以接收下一字节数据
　　0=RCREG 为空
TXIF　发送中断标志位
　　0=TXREG 寄存器为满
　　1=TXREG(发送缓冲)寄存器为空

TXIF 的重要性　要发送一个字节的数据，需要将数据写入到 TXREG 寄存器。在将字节数据送入 TXREG 寄存器时，TXIF 标志位就被清零。当整个字节通过 TX 引脚传送出去后，TXIF 标志位将变为高电平，表明它已经准备好接收下一字节的数据。所以，在写入新字节数据到 TXREG 寄存器之前要检查该标志位，否则会在发送之前将上一个字节的数据冲掉

正如在第 9 章看到的，该寄存器的几个位可用作计时器标志位。这些标志位在 PIR1 寄存器中的位置不是固定的，在以后的 PIC18 产品中可能会改变

图 10-11　PIR1(外部中断寄存器 1)

例 10-2 给出了使用 9600 波特率串行传输数据的程序。例 10-3 说明了怎样连续地传送 YES 字符串。

例 10-2　编制程序，让 PIC18 以 9600 波特率连续地串行传输字符 G。假设 XTAL=10 MHz。

解：

```
        MOVLW B'00100000'  ;enable transmit and choose low baud rate
        MOVWF TXSTA        ;write to reg
        MOVLW D'15'        ;9600 bps (Fosc / (64 * Speed) - 1)
        MOVWF SPBRG        ;write to reg
        BCF TRISC, TX      ;make TX pin of PORTC an output pin
        BSF RCSTA, SPEN    ;enable the entire serial port of PIC18
OVER    MOVLW A'G'         ;ASCII letter 'G' to be transferred
S1      BTFSS PIR1, TXIF   ;wait until the last bit is gone
        BRA S1             ;stay in loop
        MOVWF TXREG        ;load the value to be transferred
        BRA OVER           ;keep sending letter 'G'
```

401
~
402

例 10-3 编制程序，串行地传输消息 YES，使用 9600 波特率、8 位数据和 1 位结束位。一直循环该操作。

解：

```
        MOVLW B'00100000'        ;enable transmit and choose low baud
        MOVWF TXSTA              ;write to reg
        MOVLW D'15'              ;9600 bps (Fosc / (64 * Speed) - 1)
        MOVWF SPBRG              ;write to reg
        BCF TRISC, TX            ;make TX pin of PORTC an output pin
        BSF RCSTA, SPEN          ;enable the serial port
OVER    MOVLW A'Y'               ;ASCII letter 'Y' to be transferred
        CALL  TRANS
        MOVLW A'E'               ;ASCII letter 'E' to be transferred
        CALL  TRANS
        MOVLW A'S'               ;ASCII letter 'S' to be transferred
        CALL  TRANS
        MOVLW 0x0                ;NULL to purge the buffer
        CALL TRANS
        BRA   OVER               ;keep doing it
TRANS ;----serial data transfer subroutine
S1      BTFSS PIR1, TXIF         ;wait until the last bit is gone
        BRA S1                   ;stay in loop
        MOVWF TXREG              ;load the value to be transmitted
        RETURN                   ;return to caller
```

10.3.8 TXIF 标志位的重要性

为了理解 TXIF 的重要性，请看下面 PIC18 在通过 TX 传输字符时的步骤。

(1) 将要传输的字节字符写入到 TXREG 寄存器。

(2) TXIF 标志位在内部变为 1，表明 TXREG 寄存器中有一个字节数据，在发送该字节数据之前不会接收其他的字节数据。

(3) TSR(发送移位寄存器)读取 TXREG 寄存器的字节数据，开始传输字节数据的起始位。

(4) TXIF 被清零，表明上一字节正在发送，TXREG 寄存器已经准备就绪接收下一字节数据。

(5) 对于 8 位字符，每次传送一位。

(6) 查询 TXIF 标志位，确保 TXREG 寄存器不会重叠加载数据。如果在 TSR 取走上一字节数据之前向 TXREG 寄存器写入新的字节数据，那么上一个字节数据将会丢失。

从上面的讨论中可以得出结论，通过查询 TXIF 标志位，就能知道 PIC18 是否准备就绪接收下一字节数据。要查询 TXIF 标志位，可以使用指令 BTFSS PIR1,TXIF 或者使用中断，这将在第 11 章学习到。在第 11 章，将会介绍如何使用中断串行地传输数据，从而可以避免指令 BTFSS PIR1,TXIF 打断微控制器的工作。

403

10.3.9 PIC18 串行数据接收编程

对 PIC18 字节数据的串行接收进行编程，要遵循下面的步骤。

(1) 将 RXSTA 寄存器赋值为 90H，使能连续接收，并且采用 8 位数据格式。

(2) 将 TXSTA 寄存器赋值为 00H，选择低波特率。

(3) 将SPBRG寄存器赋值为表10-4中的某个值,来设定波特率(假设XTAL=10 MHz)。

(4) 将PORTC(RC7)的RX引脚用作PIC的数据输入端。

(5) 查询PIR1寄存器的RCIF标志位是否出现高电平,以判断是否接收到一个完整的字符数据。

(6) 当RCIF变为高电平时,表明RCREG寄存器里有一个字节数据。把它传送到一个安全的位置存储。

(7) 要接收下一字符,返回到步骤(5)。

例10-4给出了上面步骤的程序代码。

例10-4 对PIC18编制程序,让它串行接收数据并传送到PORTB。设波特率为9600,使用8位数据和1位结束位。

解:

```
        MOVLW B'10010000'         ;enable receive and serial port itself
        MOVWF RCSTA               ;write to reg
        MOVLW D'15'               ;9600 bps (Fosc / (64 * Speed) - 1)
        MOVWF SPBRG               ;write to reg
        BSF   TRISC, RX           ;make RX pin of PORTC an input pin
        CLRF  TRISB               ;make port B an output port
;get a byte from serial port and place it on PORTB
R1      BTFSS PIR1, RCIF          ;check for ready
        BRA   R1                  ;stay in loop
        MOVFF RCREG, PORTB        ;save value into PORTB
        BRA   R1                  ;keep doing that
```

10.3.10 RCIF标志位的重要性

在从RX引脚接收数据时,PIC18需要遵循下面的步骤。

(1) 当PIC18接收到起始位时,表明下一位数据就是它要接收的字符的首位。

(2) 对于8位字符,每次接收一位。当最后一位接收完毕时,组成一个字节数并放入RCREG。

(3) 接收结束位。在接收到结束位时,PIC18置RCIF=1,说明已经接收整个字符字节,并且在接收新的下一个字符之前必须将该接收到的字节数取走。

404

(4) 检查RCIF标志位是否变为高电平,可以判断RCREG寄存器是否接收到一个字节数。在丢失它以前,需要将它送入其他寄存器或者存储器的安全位置。

(5) 当RCREG的内容被读取(赋值)到一个安全位置时,RCIF标志位将由UART强制变回到0。这时,允许下一个接收到的字节数被放入RCREG寄存器,并且可以防止多次读取相同的字节数的情况。

从上面的讨论中可以得出结论,通过查询RCIF标志位能够判断PIC18是否准备就绪接收字符字节数。如果不能将RCREG的值复制到安全位置,那么存在丢失所接收到的字节数据的风险。更重要的是,要注意RCIF标志位是由PIC18置为高电平的,而在RCREG寄存器的数据被读取时又会被CPU清零。也要注意,如果在RCIF标志位变为高电平以前把

RCREG的内容复制到安全的位置,那么将会有复制垃圾数据的风险。RCIF标志位可以使用指令 STFSS PIR1,RCIF 或者使用中断来查询,这将在第11章中介绍。

10.3.11　PIC18的波特率翻两番

要提高PIC18数据传输的波特率,可以使用两种方法:

(1) 使用更高频率的晶振;

(2) 改变TXSTA寄存器里面的一个二进制位,如下文所示。

方法(1)对于很多情况都是不可行的,因为系统晶振是固定的。因此,只能利用方法(2)。有一个软件方法可以在保持相同的晶振频率下让PIC18的波特率翻两番。这可以由TXSTA寄存器的BRGH位完成。当PIC18上电时,TXSTA寄存器的D2位(BRGH位)为0。可以使用软件方法将它设为1,这样能获得4倍的波特率。

为了理解这种方法是如何将波特率变成4倍的,下面来介绍BRGH位(TXSTA寄存器的D2位),它的取值可以是0或者1。下面将分别加以讨论。

1. BRGH=0时的波特率

当BRGH=0时,PIC18把$F_{osc}/4$再除以16,并将得到的信号用作UART的频率来设定波特率。假设XTAL=10 MHz,则可以得到:

```
Instruction cycle freq. = 10 MHz / 4 = 2.5 kHz
and
2.5 MHz / 16 = 156,250 Hz because BRGH = 0
```

这就是UART用来设定波特率的频率。这是目前介绍的所有例子的基础,因为它是PIC18上电时的默认值。在BRGH=0时的波特率已经列在表10-4和表10-5中了。

405

2. BRGH=1时的波特率

在固定的晶振频率下,把BRGH置为1,可以得到4倍的波特率。当BRGH位(TXSTA寄存器的D2)被置为1时,XTAL的$F_{osc}/4$将被再除以4(而不是16),于是UART使用这个频率来设定波特率。假设XTAL=10 MHz,则可以得到:

```
Instruction cycle freq. = 10 MHz / 4 = 2.5 MHz
and
2.5 MHz / 4 = 62500 Hz because BRGH = 1
```

这就是在BRGH=1时,UART用来设定波特率的频率。

表10-8列出了不同条件下将相同的值送入SPBREG的情况;不过,在BRGH=1时,波特率将被提高为原来的4倍。请看例10-5到例10-7,验证表10-6和表10-7中给出的数据。

表10-6　不同波特率对应的SPBRE值(F_{osc}=10 MHz,BRGH=1)

波特率	SPBRG(十进制)	SPBRG(十六进制)
57 600	10	0A
38 400	15	0F
19 200	32	20
9 600	64	40
4 800	129	81

注意:对于F_{osc}=10 MHz,有SPBRG=(625 000/波特率)−1。

例 10-5 试确定满足下面波特率的 SPBRG 值。

(a)当 BRGH=1 时使用波特率 9600　　(b)当 BRGH=1 时使用波特率 4800

假设 XTAL =10 MHz。

解：

因为 XTAL=10 MHz，所以 $F_{osc}/4=2.5$ MHz。因为 BRGH=1，所以有 UART 频率=2.5 MHz/4=625 000 Hz。

(a)(625 000/9600)−1=64，所以 SPBRG=64 或者 SPBRG=40H(十六进制)。

(b)(625 000/4800)−1=129，所以 SPBRG=129 或者 SPBRG=81H(十六进制)。



406

表 10-7　不同波特率对应的 SPBRG 值(XTAL=10 MHz)

	BRGH=0	BRGH=1
波特率	SPBRG(十进制)	SPBRG(十进制)
57 600	2	10
38 400	3	15
19 200	7	32
9 600	15	64
4 800	32	129

SPBRG=(156 250/波特率)−1　　SPBRG=(62 500/波特率)−1

表 10-8　SPBRG 值和波特率(BRGH=0 和 BRGH=1)(XTAL=10 MHz)

	BRGH = 0	BRGH = 1
SPBRG(十进制)	波特率	波特率
15	9600	38 400
32	4800	19 200
64	2400	9 600

表 10-9　不同波特率对应的 SPBRG 值(XTAL=4 MHz)

	BRGH=0	BRGH = 1
波特率	SPBRG(十进制)	SPBRG(十进制)
19 200	3	12
9 600	6	25
4 800	12	51
2 400	25	103

SPBRG=(62 500/波特率)−1　　SPBRG=(250 000/波特率)−1

例 10-6 编制程序，让 PIC18 以 57 600 的波特率连续地串行发送字符 G。假设 XTAL＝10 MHz。使用 BRGH＝1 模式。

解：

```
       MOVLW B'00100100' ;enable transmit and choose high baud rate
       MOVWF TXSTA       ;write to reg
       MOVLW D'10'       ;57600 bps (Fosc / (16 * Speed) - 1)
       MOVWF SPBRG       ;write to reg
       BCF TRISC, TX     ;make TX pin of PORTC an output pin
       BSF RCSTA, SPEN   ;enable the entire Serial port of PIC18
OVER   MOVLW A'G'        ;ASCII letter 'G' to be transferred
S1     BTFSS PIR1, TXIF  ;wait until the last bit is gone
       BRA S1            ;stay in loop
       MOVWF TXREG       ;load the value to be transferred
       BRA OVER          ;keep sending letter 'G'
```

407

10.3.12 波特率的误差计算

在计算波特率的时候，采用的是整数作为 SPBRG 寄存器的值，因为 PIC 微控制器只能使用整数。当去掉计算值的小数部分时，波特率将可能存在误差。有几种方法可以用来计算这种误差。其中一种方法就是采用下面的公式：

误差＝(SPBRG 的计算值－整数部分)/整数部分

例如，假设 XTAL＝10 MHz，BRGH＝0，波特率为 9600，则可以得到：

SPBRG 的值＝(156 250/9600)－1＝16.27－1＝15.27＝15

其误差为

(15.27－15)/16＝1.7%

另一种计算误差率的方法是：

误差＝(计算的波特率－期望波特率)/期望波特率

其中期望波特率由 $X=[(F_{osc}/\text{期望波特率})64]-1$ 可以算出，而整数 X(送入 SPBRG 寄存器的值)可用于计算波特率，计算公式如下：

待计算的波特率＝$F_{osc}/[64(X+1)]$(当 BRGH＝0 时)

当 XTAL＝10 MHz、波特率为 9600 时，可以计算得到 $X=15$。因此，得到待计算的波特率＝10 MHz/[64(15＋1)]＝9765。现在计算误差如下：

误差＝(9765－9600)/9600＝1.7%

这和过去使用其他方法计算得到的值是一样的。表 10-10 和表 10-11 分别给出了在 10 MHz 和 4 MHz 晶振下的 SPBRG 值的误差。比较例 10-7 和例 10-8，观察如何使用两种不同的方法来计算误差。

例 10-7 假设 XTAL＝10 MHz，计算下面波特率的误差：

(a) 2400　　(b) 1200　　(c) 19 200　　(d) 57 600

使用 BRGH＝0 模式。

解：

(a) SPBRG 值=(156 250/2400)−1=65.1−1=64.1=64

误差=(64.1−64)/65=0.15%

(b) SPBRG 值=(156 250/1200)−1=130.2−1=129.2=129

误差=(129.2−129)/130=0.15%

(c) SPBRG 值=(156 250/19 200)−1=8.138−1=7.138=7

误差=(7.138−7)/8=1.7%

(d) SPBRG 值=(156 250/57 600)−1=2.71−1=1.71=1

误差=(1.71−1)/2=35%

该误码率太高。将计算出来的数值取整，结果会是如何呢?

误差=(3−2.7)/3=10%，也就是说，使用 SPBRG=2 代替 SPBRG=1。

例 10-8 假设 XTAL=10 MHz，计算下面波特率的误差：

(a) 2400 (b) 1200

使用 BRGH=0 模式。

解：

(a) SPBRG 值=(156 250/2400)−1=65.1−1=64.1=64

计算的波特率是 156 250/(64+1)=2403

误差=(2403−2400)/2400=0.12%

(b) SPBRG 值=(156 250/1200)−1=130.2−1=129.2=129

计算的波特率是 156 250/(129+1)=1202

误差=(1202−1200)/1200=0.16%

表 10-10 不同波特率对应的 SPBRG 值(XTAL=10 MHz)

	BRGH =0		BRGH =1	
波特率	SPBRG	误差	SPBRG	误差
38 400	3	1.5%	15	1.7%
19 200	7	1.7%	32	1.3%
9 600	15	1.7%	64	0.15%
4 800	32	1.3%	129	0.15%

SPBRG=(156 250/波特率)−1 SPBRG=(62 500/波特率)−1

表 10-11 不同波特率对应的 SPBRG 值(XTAL=4 MHz)

	BRGH = 0		BRGH = 1	
波特率	SPBRG	误差	SPBRG	误差
19 200	2	8.3%	12	0.15%
9 600	6	8%	25	0.15%
4 800	12	0.15%	51	0.15%
2 400	25	0.16%	103	0.16%

SPBRG=(62 500/波特率)−1 SPBRG=(250 000/波特率)−1

408~409

请看下面的一些例子，掌握 PIC18 串行端口的编程要领。

例 10-9 假设将一个开关连接到引脚 RD7。编制程序，监视该开关的状态，并且根据开关状态将如下的消息送入串行端口：

SW=0 送 NO

SW=1 送 YES

假设 XTAL=10 MHz，波特率为 9600。

解：

```
        BSF TRISD,7         ;PORTD.7 as in input for SW
        MOVLW 0x20          ;enable transmit and choose low baud rate
        MOVWF TXSTA         ;write to reg
        MOVLW D'15'         ;9600 bps (Fosc / (64 * Speed) - 1)
        MOVWF SPBRG         ;write to reg
        BCF   TRISC, TX     ;make TX pin of PORTC an output pin
        BSF   RCSTA, SPEN   ;enable the entire serial port of PIC18
OVER    BTFSS PORTD,7
        BRA   NEXT
        MOVLW high(MESS1)   ;if SW = 0 display "NO"
        MOVWF TBLPTRH
        MOVLW low(MESS1)
        MOVWF TBLPTRL
FN      TBLRD*+             ;read the character
        MOVF  TABLAT,W
        BZ    EXIT          ;check for end of line
        CALL  SENDCOM       ;send character to serial port
        BRA   FN            ;repeat
NEXT    MOVLW high(MESS2)   ;if SW = 1 display "YES"
        MOVWF TBLPTRH
        MOVLW low(MESS2)
        MOVWF TBLPTRL
LN      TBLRD*+             ;read the character
        MOVF  TABLAT,W      ;Z = 1 if NULL
        BZ    EXIT          ;check for end of line
        CALL  SENDCOM       ;send character to serial port
        BRA   LN            ;repeat
EXIT    MOVLW 0x20          ;send space
        CALL  SENDCOM
        GOTO  OVER
;-------------
SENDCOM
S1      BTFSS PIR1, TXIF    ;wait until the last bit is gone
        BRA   S1            ;stay in loop
        MOVWF TXREG         ;load the value to be transferred
        RETURN              ;return to caller
;------------------
MESS1 DB    "NO",0
MESS2 DB    "YES",0
```


例 10-10 编制程序，连续地发送消息 The Earth is but One Country 到串行端口。假设 SW 连接到引脚 RB0。监视它的状态，并且按下面的波特率发送：

SW=0，9600 波特率

SW=1，38 400 波特率

假设 XTAL=10 MHz。

解：如表 10-8 所示，通过改变 TXSTA 寄存器的 BRGH 位可以得到 4 倍的波特率。

```
      BSF    TRISB,0      ;PORTB.0 as in input for SW
      BCF    TRISC, TX    ;make TX pin of PORTC an output pin
      BSF    RCSTA, SPEN  ;enable the entire serial port of PIC18
      MOVLW  0x20         ;transmit at low baud rate
      MOVWF  TXSTA        ;write to reg
      MOVLW  D'15'        ;9600 bps (Fosc / (64 * Speed) - 1)
      MOVWF  SPBRG        ;write to reg
OVER  BTFSC  PORTB,0      ;test bit PORTB.0 and skip if LOW
      BSF    TXSTA,BRGH   ;transmit at high rate by making BRGH = 1
      MOVLW  upper(MESSAGE)
      MOVWF  TBLPTRU
      MOVLW  high(MESSAGE)
      MOVWF  TBLPTRH
      MOVLW  low(MESSAGE)
      MOVWF  TBLPTRL
NEXT  TBLRD*+             ;read the character
      MOVF   TABLAT,W     ;place it in WREG
      BZ     OVER         ;if end of line, start over
      CALL   SENDCOM      ;send char to serial port
      BRA    NEXT         ;repeat for the next character
;-------------
SENDCOM
S1    BTFSS  PIR1, TXIF   ;wait until the last bit is gone
      BRA    S1           ;stay in loop
      MOVWF  TXREG        ;load the value to be transmitted
      RETURN              ;return to caller
;------------------
MESSAGE DB  "The Earth is but One Country",0
```

411

10.3.13 发送和接收

假设 PIC18 串行端口连接到 IBM PC 的 COM 端口，使用 PC 上的 HyperTerminal 程序发送和接收串行数据。PIC18 的端口 *B* 和端口 *D* 分别连接到 LED 和开关。程序10-1给出了具有以下功能的 PIC18 程序：(a)发送一次消息 YES 到 PC 屏幕；(b)从开关读取数据，并通过串行端口送到 PC 屏幕；(c)接收 HyperTerminal 的按键输入，并传送给 LED。程序执行(a)一次，而不断地执行(b)和(c)。在 XTAL=10 MHz 时，使用 9600 的波特率。

程序 10-1 发送和接收

```
;Program 10-1 Transmit and Receive
      ORG  0
  ;initialize the serial ports for both transmit and receive
      MOVLW  B'00100100'        ;enable transmit, choose high baud
      MOVWF  TXSTA              ;write to reg
      MOVLW  B'10010000'        ;enable receive, serial port itself
      MOVWF  RCSTA              ;write to reg
      MOVLW  D'15'              ;9600 bps (Fosc / (64 * Speed) - 1)
      MOVWF  SPBRG              ;write to reg
      BSF    RCSTA, SPEN        ;enable the serial port itself
      BCF    TRISC, TX          ;make TX pin of PORTC an output
      BSF    TRISC, RX          ;make RX pin of PORTC an input
      CLRF   TRISB              ;make port B an output port
      SETF   TRISD              ;make port D an input port
  ;send the message "YES"
```

```
        MOVLW 'Y'                  ;ASCII letter 'Y' to be transferred
        CALL  TRANS
        MOVLW 'E'                  ;ASCII letter 'E' to be transferred
        CALL  TRANS
        MOVLW 'S'                  ;ASCII letter 'S' to be transferred
        CALL  TRANS
    ;get a byte from switches and transmit data to PC screen
OVER    MOVF PORTD,W               ;get a byte from SW of PORTD
        CALL  TRANS                ;transmit it via serial port
    ;as keys are pressed on PC receive data and put it on LEDs
        CALL  RECV                 ;receive the byte from serial port
        MOVWF PORTB                ;display it on LEDS of PORTB
        BRA   OVER                 ;keep doing it
;--serial transfer (WREG needs the byte to be transmitted)
TRANS
S1      BTFSS PIR1, TXIF           ;wait until the last bit is gone
        BRA   S1                   ;
        MOVWF TXREG                ;load the value to be transferred
        RETURN                     ;return to caller
;----serial data receive subroutine (WREG = received byte)
RECV    BTFSS PIR1, RCIF            ;check for ready
        BRA   RECV                  ;stay in loop
        MOVF RCREG,W                ;save value in WREG
        RETURN
```

412

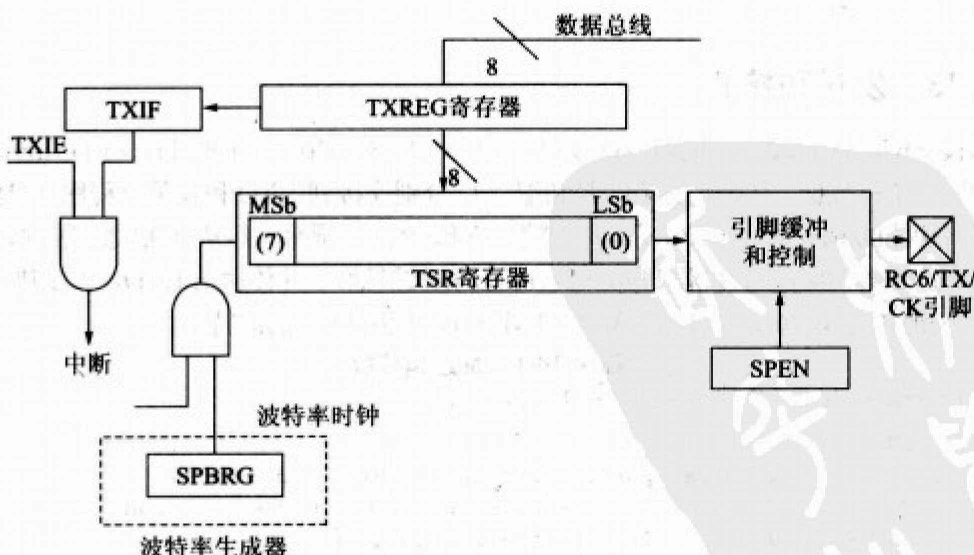


图 10-12　USART 发送的简化方框图

10.3.14　基于中断的数据传输

到现在为止，读者可能已经注意到查询 TXIF 和 RXIF 标志位是很浪费微控制器时间的。为了避免浪费微控制器的时间，可以使用中断来代替查询。第 11 章将会介绍如何使用中断对 PIC18 串行通信端口编程。

10.3.15 复习题

1. PIC18 的哪个寄存器是用来设置波特率的?
2. 如果 XTAL=10 MHz,那么 URAT 用来设置波特率的频率是多少?(假设使用默认模式)。
3. TXSTA 寄存器的哪一位是用来选择低/高波特率的?
4. 当 XTAL=10 MHz 时,要产生 9600 波特率,SPBRG 的值应为多少?结果分别用十进制和十六进制表示。
5. 要串行发送一字节数据,必须将数据放在寄存器________中。
6. TXSTA 表示________,它是一个________位的寄存器。
7. 哪个寄存器是用来设置数据帧大小的?
8. 判断对错:TXSTA 是一个位可寻址的寄存器。
9. TXIF 在什么时候变高?又是在什么时候清零?
10. BRGH 位在哪个寄存器里?当 PIC18 上电时,它的状态是什么?

10.4 PIC18 串行端口的 C 编程

本节将介绍 PIC18 芯片串行端口的 C 语言编程。

10.4.1 PIC18 C 的数据发送和接收

如在第 7 章所看到的,PIC18 的所有 SFR 都可以由 C18 编译器使用适当的头文件来直接访问。例 10-11 到例 10-15 介绍了如何使用 PIC18 C 对串行端口编程。可以将 PIC18 Trainer 连接到 PC 的 COM 端口,并使用 HyperTerminal 程序来测试这些例子。注意,从例 10-11 到例 10-15 都是上一节中汇编程序的 C 语言版本。

例 10-11 编制 C 程序,让 PIC18 以 9600 波特率连续地串行传输字符 G。使用 8 位数据和 1 位结束位。假设 XTAL=10 MHz。

解:

```
#include <P18F4580.h>
void main(void)
  {
    TXSTA=0x20;            //choose low baud rate,8-bit
    SPBRG=15;              //9600 baud rate/ XTAL = 10 MHz
    TXSTAbits.TXEN=1;
    RCSTAbits.SPEN=1;

    while(1)
      {
        TXREG='G';         //place value in buffer
        while(PIR1bits.TXIF==0);      //wait until all gone
      }
  }
```

10.4.2　复习题

1. 判断对错：PIC18 的所有 SFR 都可以在 C18 C 编译器中访问。
2. 判断对错：C18 编译器支持 PIC18 的位可寻址寄存器。
3. 判断对错：TXIF 标志位在向 TXREG 寄存器写入一个字符的时候清零。
4. 哪个寄存器是用来设定波特率的？

414

5. BRGH 位属于哪个寄存器，它的作用是什么？

例 10-12　编制 PIC18 的 C 程序，以 9600 的波特率串行传输消息 YES，使用 8 位数据和 1 位结束位。一直循环。

解：

```
#include <P18F458.h>
void SerTx(unsigned char);
void main(void)
  {
    TXSTA=0x20;              //choose low baud rate,8-bit
    SPBRG=15;                //9600 baud rate/ XTAL = 10 MHz
    TXSTAbits.TXEN=1;
    RCSTAbits.SPEN=1;
    while(1)
      {
        SerTx('Y');
        SerTx('E');
        SerTx('S');
      }
  }
void SerTx(unsigned char c)
  {
    while(PIR1bits.TXIF==0);     //wait until transmitted
    TXREG=c;                     //place character in buffer
  }
```

例 10-13　编制 PIC18 的 C 程序，接收串行数据并传送到端口 *B*。设波特率为 9600，使用 8 位数据和 1 位结束位。

解：

```
#include <P18F458.h>
void main (void)
  {
    TRISB = 0;                   //PORTB an output
    RCSTA=0x90;                  //enable serial port and receiver
    SPBRG=15;                    //9600 baud rate/ XTAL = 10 MHz
    while(1)                     //repeat forever
      {
        while(PIR1bits.RCIF==0); //wait to receive
        PORTB=RCREG;             //save value
      }
  }
```

415

例 10-14 编制 C18 程序，向串行端口发送两个不同的字符串。假设 SW 连接到引脚 PORTB.5。监视它的状态，并且根据如下的逻辑执行操作：

SW=0：发送你的名

SW=1：发送你的姓

假设 XTAL=10 MHz，波特率为 9600，使用 8 位数据。

解：

```
#include <P18F458.h>
#define MYSW PORTBbits.RB5        //INPUT SWITCH
void main(void)
  {
    unsigned char z;
    unsigned char fname[]="ALI";
    unsigned char lname[]="SMITH";
    TRISBbits.TRISB5 = 1;  //an input
    TXSTA=0x20;            //choose low baud rate, 8-bit
    SPBRG=15;              //9600 baud rate/ XTAL = 10 MHz
    TXSTAbits.TXEN=1;
    RCSTAbits.SPEN=1;
    if(MYSW==0)                           //check switch
      {
        for(z=0;z<3;z++)                  //write name
          {
            while(PIR1bits.TXIF==0); //wait for transmit
            TXREG=fname[z];               //place char in buffer
          }
      }
    else
      {
        for(z=0;z<5;z++)                  //write name
          {
            while(PIR1bits.TXIF==0); //wait for transmit
            TXREG=lname[z];               //place value in buffer
          }
      }
     while(1);
  }
```

416

例 10-15 编制 PIC18 的 C 程序，发送两条消息 Normal Speed 和 High Speed 到串行端口。假设 SW 连接到引脚 PORTB.0。监视它的状态，并且按下列要求来设置波特率：

SW=0：9600 波特率

SW=1：38 400 波特率

假设两种情况都使用 XTAL=10 MHz。

解：

```
#include <P18F458.h>
#define MYSW PORTBbits.RB5        //INPUT SWITCH
void main(void)
  {
```

```
    unsigned char z;
    unsigned char Mess1[]="Normal Speed";
    unsigned char Mess2[]="High Speed";
    TRISBbits.TRISB5 = 1;  //an input
    TXSTA=0x20;            //choose low baud rate, 8-bit
    SPBRG=15;              //9600 baud rate/ XTAL = 10 MHz
    TXSTAbits.TXEN=1;
    RCSTAbits.SPEN=1;
    if(MYSW==0)
      {
        for(z=0;z<12;z++)
          {
            while(PIR1bits.TXIF==0); //wait for transmit
            TXREG=Mess1[z];          //place value in buffer
          }
      }
    else
      {
        TXSTA=TXSTA|0x4;             //for high speed
        for(z=0;z<10;z++)
          {
            while(PIR1bits.TXIF==0); //wait for transmit
            TXREG=Mess2[z];          //place value in buffer
          }
      }
      while(1);
  }
```

417

小结

本章首先介绍了串行通信的基础知识。串行通信(也就是每次只发送一位数据)可用于远距离的数据传送,因为在并行通信方式下长距离发送一个或更多字节时会出现数据失真。串行通信的另一个优点就是可以在电话线上传输。实现串行通信的方法有两种:同步和异步。在同步通信中,数据按字节分组传送;而在异步通信中,每次发送一个字节数据。数据通信可以是单工(只能发送但不能接收)、半双工(可发送可接收,但不能同时进行)或者是全双工(可同时发送和接收)。RS232 是串行通信连接器的一个标准。

本章还讨论了 PIC18 的 UART,介绍了 PIC18 和 RS232 连接器的接口技术,以及如何改变 PIC18 的波特率。另外,还讨论了 PIC18 串行通信的特性和编程。最后还介绍了如何使用汇编和 C 语言对 PIC18 芯片的串行端口编程。

习题

1. 在并行数据传输和串行数据传输中,哪一种成本更高?
2. 判断对错:0-V 和 5-V 的数字脉冲不需转换(调制)就可以在电话中传输。

3. 写出字符 Z 的 ASCII 码(0101 1010)的数据帧,不带奇偶校验,使用 1 位结束位。
4. 如果没有数据传输,导线为高电平,这叫作________(标记,空)。
5. 判断对错:结束位可以是 1,2 甚至没有。
6. 如果数据大小是 7,1 位结束位,无奇偶校验,请计算额外开销所占的百分比。
7. 判断对错:RS232 电压是 TTL 兼容的。
8. MAX232 芯片的功能是什么?
9. 判断对错:DB-25 和 DB-9 的前 9 个引脚都是兼容的。
10. RS232 有多少个引脚用于 IBM 串行电缆,为什么?
11. 判断对错:电缆越长,数据传输波特率越高。
12. 指出两个 PC 间串行传输需要的最少信号数。它们是什么?
13. 如果两台 PC 通过 RS 232 而不是调制解调器连接在一起,那么这两台 PC 都是________(DTE,DCE)至________(DTE,DCE)连接。
14. 指出 RS232 的 9 个最重要的信号。
15. 如果要用异步数据传输方式传送 200 页的 ASCII 数据,一共需要传送多少位数据?假设数据大小是 8 位,1 位结束位,无奇偶校验。每一页有 80×85 个字符。
16. 在 15 题中,如果使用 9600 的波特率,需要的传输时间是多久?

418

17. MAX232DIP 的封装有________个引脚。
18. 指出 MAX232 的 V_{CC} 和 GND 引脚。
19. MAX233DIP 的封装有________个引脚。
20. 指出 MAX233 的 V_{CC} 和 GND 引脚。
21. MAX232 和 MAX233 的引脚是否兼容?
22. MAX232 和 MAX233 各有什么优点和缺点?
23. MAX232/233 带有________个 RX 线驱动器。
24. MAX232/233 带有________个 TX 线驱动器。
25. 指出 PIC18 的 TX 和 RX 引脚是怎样通过 MAX232 第二套线驱动器连接到 DB-9 RS232 连接器的。
26. 指出 PIC18 的 TX 和 RX 引脚是怎样通过 MAX233 第二套线驱动器连接到 DB-9 RS232 连接器的。
27. MAX233 相比 MAX232 芯片的优势在哪里?
28. PIC18 的哪些引脚是用作串行通信的,它们的功能是什么?
29. PC 的 HyperTerminal 程序支持下面的哪些波特率?
    (a)4800 (b)3600 (c)9600 (d)1800 (e)1200 (f)19 200
30. PIC18 的哪个定时器可用于波特率编程?
31. TXSTA 的哪一位用于设置波特率速度?
32. TXREG 寄存器在串行数据通信中的作用是什么?
33. TXREG 是一个________位的寄存器。
34. TXSTA 寄存器在串行数据通信中的作用是什么?
35. TXSTA 是一个________位的寄存器。
36. 假设 XTAL=10 MHz,计算下面各个波特率的 SPBRG 值(十进制和十六进制)。
    (a)9600 (b)4800 (c)1200
37. 如果使用 SPBRG=15 来编程,波特率是多少?假设 XTAL=10 MHz。
38. 编制 PIC18 程序,以 1200 的波特率连续地串行传输字符 Z。假设 XTAL=10 MHz。
39. 编制 PIC18 程序,以 57 600 的波特率连续地串行传输消息 The earth is but one country and mankind

its citizens。假设 XTAL=10 MHz。

40. TXIF 标志位什么时候会变为高电平并清零?
41. RCIF 标志位什么时候会变为高电平并清零?
42. TXIF 和 RCIF 属于哪个寄存器? 这个寄存器可以位可寻址吗?
43. RCSTA 寄存器的 SPEN 位的作用是什么?
44. 对于不能接收任何串行数据的情况,你如何使用一条指令来阻止接收?
419
45. BRGH 属于哪个寄存器? 它对数据传输速率的作用是什么?
46. 当 PIC18 上电时,BRGH 位是高电平还是低电平?
47. 计算在 XTAL=16 MHz、BRGH=0 时下面各个波特率对应的 SPBRG 值。
    (a)9600　(b)19 200　(c)38 400　(d)57 600
48. 计算在 XTAL=16 MHz、BRGH=1 时下面各个波特率对应的 SPBRG 值。
    (a)9600　(b)19 200　(c)38 400　(d)57 600
49. 计算在 XTAL=20 MHz、BRGH=0 时下面各个波特率对应的 SPBRG 值。
    (a)9600　(b)19 200　(c)38 400　(d)57 600
50. 计算在 XTAL=20 MHz、BRGH=1 时下面各个波特率对应的 SPBRG 值。
    (a)9600　(b)19 200　(c)38 400　(d)57 600
51. 计算第 47 题中的波特率误差。
52. 计算第 48 题中的波特率误差。
53. 编制 PIC18 的 C 程序,以 1200 的波特率连续地串行传输字符 Z。
54. 编制 PIC18 的 C 程序,以 57 600 的波特率连续地串行传输消息 The earth is but one country and mankind its citizens。

复习题答案

10.1 节

1. 更快,更贵　2. 错误。这是单工。　3. 正确。　4. 异步
5. 对于二进制数 0100 0101,串行传输的位序列为:
   (a) 0 (起始位)　(b) 1　(c) 0　(d) 1　(e) 0　(f) 0　(g) 0　(h) 1　(i) 0　(j) 1(结束位)
6. 2 位(一个是起始位,另一个是结束位)。因此,对于 8 位字符,共有 10 位要传输。
420
7. 一共传输 10 000×10=100 000 位。100 000/9600=10.4s;2/10=20%。
8. 正确。　9. +3V～+25V　10. 正确。　11. 1　12. COM1,COM2

10.2 节

1. 正确。　2. 引脚 RC6 和 RC7。引脚 RC6 用作 TX,而引脚 RC7 用作 RX。
3. 它们用来把 RS232 电压转换成 TTL 电平,反之亦然。
4. 2,2　5. 它不需要 MAX232 的 4 个电容。

10.3 节

1. SPBRG　2. 156 250Hz　3. BRGH　4. 十进制的 15(或者十六进制的 F),因为 156 250/9600−1=15。
5. TXREG　6. 发送状态和控制寄存器,8　7. TXSTA　8. 正确。
9. 在传输结束位的时候,它变为高电平。当向 TXREG 写入要传送的字节数据时,它将被清零。

10. TXSTA；在上电时它为低电平。

10.4 节

1. 正确。　2. 正确。　3. 正确。　4. SPBRG

5. TXSTA。它允许在相同的晶振下得到 4 倍的波特率。

第 11 章 用汇编和 C 语言进行中断编程

学习目标：

- ☐ 中断和查询的比较
- ☐ ISR(中断服务程序)的作用
- ☐ PIC18 主要的中断类型
- ☐ 中断向量表的作用
- ☐ PIC18 的中断使能与禁止
- ☐ 使用中断的 PIC18 定时器编程
- ☐ PIC18 的外部硬件中断
- ☐ PIC18 基于中断的串行通信编程
- ☐ PIC18 的中断优先级
- ☐ PIC 中断的 C 语言编程

本章将会介绍中断的概念以及中断编程。11.1 节将讨论 PIC18 中断的基础知识。11.2 节将会介绍定时器的中断。外部硬件中断将在 11.3 节中讨论，而关于串行通信的中断则在 11.4 节中介绍。11.5 节将介绍 PORTB 有关的中断。11.6 节将讨论中断优先级。在这一章中，所有的例子都将以汇编语言和 C 语言的形式出现。

11.1 PIC18 中断

本节首先将讨论查询和中断的区别，然后再介绍 PIC18 的不同中断类型。

11.1.1 中断和查询

单个微控制器可以服务几个设备。微控制器提供服务的方法有两种：中断和查询。使用中断的方法，就是每当设备需要微控制器的服务时，设备就以发送中断信号的形式通知微控制器。当接收到中断信号时，微控制器就停止当前工作来处理设备服务。和中断相关的程序就叫作中断服务程序(ISR)或者中断处理程序。当使用查询的方法时，微控制器会不断地查询设备的状态；当满足一定状态条件时，它就提供服务。然后，它继续查询下一个设备，直到服务完每一个设备为止。虽然查询方法可以监视几个设备的状态，并且在满足一定条件时处理每一个设备，但是对于微控制器来说它不是一个高效的方法。中断的优点在于微控制器可以对许多设备提供服务(当然不是对所有设备同时提供服务)；每个设备都可以根据其优先级得

到微控制器的关注。查询方法不存在优先级的分配问题,因为它是轮流地检查每个设备的。更重要的是,微控制器使用中断方法可以忽略(屏蔽)某个设备的服务请求。而查询方法无法实现这一功能。优先选择中断方法的最重要原因是,查询方法在不需要服务的设备查询上浪费了大量的微控制器时间。因此,中断方法可以避免微控制器时间的浪费。例如,在第 9 章中讨论定时器时所使用的位查询指令 BTFSS TMR0IF,将一直等待直到定时器复位为零,而在等待的过程中微控制器是不能执行其他任务的。这样,就浪费了原本可以用来做有用工作的微控制器时间。就定时器来讲,如果使用中断方法,那么微控制器是可以执行其他任务的。无论何时 TMR0IF 标志位变为高电平,定时器都会中断微控制器的当前工作。

11.1.2 中断服务程序

对于每个中断,都需要一个中断服务程序(ISR)(或者称为中断处理程序)。当一个中断被触发时,微控制器就会运行中断服务程序。通常,对于大多数的微处理器,每个中断的地址在存储器中都有固定的位置,用来保存 ISR 地址。为 ISR 地址保留的存储器空间又被称作中断向量表。对于 PIC18,只有两个中断向量表地址,即地址 0008 和地址 0018,如表 11-1 所示。在 11.6 节讨论中断优先级时,将会讨论它们之间的区别。

424

表 11-1 PIC18 的中断向量表

中 断	ROM 地址(十六进制)
上电复位	0000
高优先级中断	0008(上电复位时的默认值)
低优先级中断	0018(请参阅 11.6 节)

11.1.3 中断执行的步骤

当一个中断被触发时,微控制器会执行以下的步骤。

(1) 微控制器完成当前指令的执行,并把下一条要执行的指令地址(程序计数器)保存到栈中。

(2) 微控制器跳转到存储器的固定地址(即中断向量表)。中断向量表引导微控制器去访问中断服务程序(ISR)的地址。

(3) 微控制器从中断向量表得到 ISR 的地址并跳转到该地址。它开始执行中断服务子例程,直到子例程的最后一条指令,即 RETFIE 指令(从中断出口处返回)。

(4) 在执行 RETFIE 指令时,微控制器返回到被中断的地方。首先,它从栈弹出栈顶字节到 PC,得到程序计数器的值(PC)。然后从该地址开始继续执行。

注意第(4)步中栈的重要性。因此,在 ISR 中必须小心地处理栈的内容。尤其要注意,ISR 同任何 CALL 子例程一样,压栈和出栈操作的数量要相同。

11.1.4 PIC18 的中断源

PIC18 有很多的中断源,其数量取决于芯片所连接的外部设备。下面是 PIC18 最普遍使用的中断源。

(1) 用于定时器的中断源,如定时器 0、定时器 1、定时器 2 等。请参阅 11.2 节。

(2) 用于外部硬件中断的 3 个中断源。引脚 RB0(PORTB.0)、RB1(PORTB.1)和 RB2

(PORTB.2),分别用于外部硬件中断 INT0、INT1 和 INT2。请参阅 11.3 节。

(3) 用于串行通信 USART 的 2 个中断源,一个中断源用于接收,另一个中断源用于发送。请参阅 11.4 节。

(4) 用于 PORTB 变化的中断源。请参阅 11.5 节。

(5) ADC(模数转换器)中断源。请参阅第 13 章。

425

(6) CCP(比较捕获脉冲宽度调制)中断源。请参阅第 15 章和第 17 章。

PIC18 的中断源要比上面所列出的更多。当在本书中学习到 PIC18 的外部设备时就会涉及这些中断源。注意,在表 11-1 中,高优先级中断的数量是有限的。例如,从地址 0008 到 000017H 的总共 8B 是用作高优先级中断的。通常,一个中断服务程序都会因为太长而无法放入该存储空间。基于这个原因,通常在向量表中放置 GOTO 指令来指向 ISR 地址。分配给中断的其余字节空间是没有使用的。在下一节中,将会给出更多中断程序的例子来澄清这些概念。

从表 11-1 可以注意到,只有 8B 的 ROM 空间被分配给复位引脚,分别是 ROM 地址 0~7。因此,在程序中需要把 GOTO 作为第一条指令,以引导处理器跳出中断向量表,如图 11-1 所示。在下一节的例子中,将会看到它是怎样工作的。

```
          ORG  0    ;wake-up ROM reset location
          GOTO MAIN ;bypass interrupt vector table

;---- the wake-up program
          ORG  100H
MAIN:     .... ;enable interrupt flags
          ....
          END
```

图 11-1 在上电时重定向 PIC18 跳出中断向量表

11.1.5 中断的使能和禁用

在复位时,所有的中断都是被禁用(屏蔽)的,也就是说当它们被触发时,微控制器不会响应。要使得微控制器能响应它们,必须使用软件方法来使能(不屏蔽)中断。INTCON(中断控制)寄存器的 D7 位就是用于使能和禁用全局中断的。图 11-3 描述了 INTCON 寄存器。GIE 位可以方便地用来禁用全部的中断。使用一条简单的指令(BCF INTCON, GIE),就可以在执行一个重要任务时完成置 GIE=0 的工作。如图 11-2所示。

426

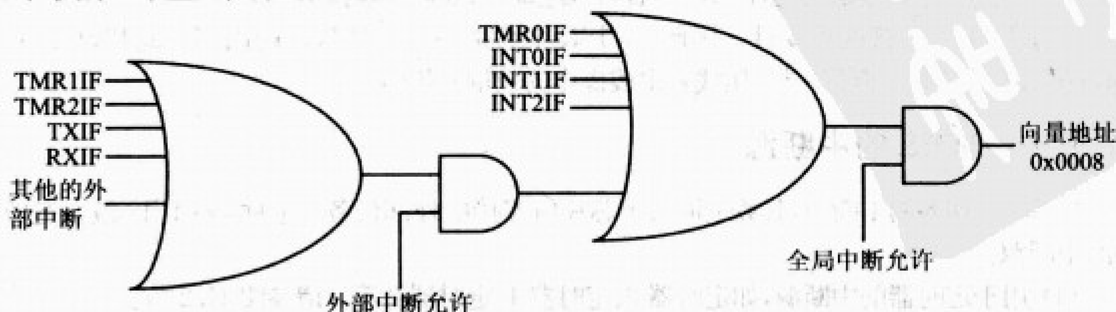
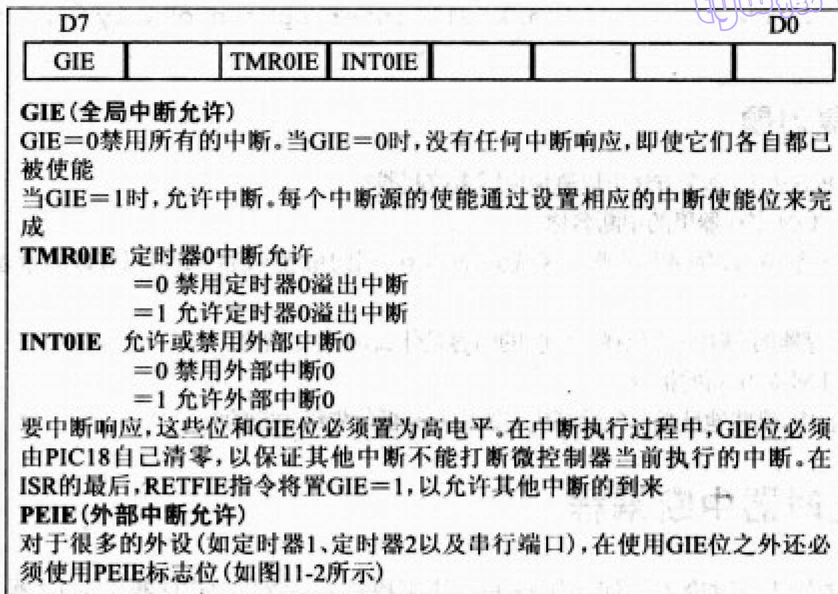


图 11-2 中断的简化示意图(上电复位时的默认状态)


D7 D0

GIE		TMR0IE	INT0IE				

GIE(全局中断允许)
GIE=0禁用所有的中断。当GIE=0时，没有任何中断响应，即使它们各自都已被使能
当GIE=1时，允许中断。每个中断源的使能通过设置相应的中断使能位来完成
TMR0IE 定时器0中断允许
=0 禁用定时器0溢出中断
=1 允许定时器0溢出中断
INT0IE 允许或禁用外部中断0
=0 禁用外部中断0
=1 允许外部中断0
要中断响应，这些位和GIE位必须置为高电平。在中断执行过程中，GIE位必须由PIC18自己清零，以保证其他中断不能打断微控制器当前执行的中断。在ISR的最后，RETFIE指令将置GIE=1，以允许其他中断的到来
PEIE(外部中断允许)
对于很多的外设(如定时器1、定时器2以及串行端口)，在使用GIE位之外还必须使用PEIE标志位(如图11-2所示)

图 11-3 INTCON(中断控制)寄存器

11.1.6 使能中断的步骤

要使能任何一个中断，必须遵循以下的步骤。

(1) 必须将 INTCON 的 D7 位(GIE)置为高电平，允许中断。可以使用 BSF INTCON, GIE 指令来实现。

(2) 当 GIE=1 时，可以通过对中断允许(IE)标志位置 1 来允许中断。因为 PIC18 有非常多的中断，所以需要很多的内含中断使能位的寄存器。图 11-2 描述了 INTCON 中用于定时器 0 的中断允许位(TMR0IE)和用于外部中断 0 的中断允许位(INT0IE)。在本书中学习外部设备时，将会介绍包括中断使能位的寄存器。要注意，如果 GIE=0，即使相应的中断使能位为高电平，也不会发生中断响应。为更好地理解这一点，请参阅例 11-1。

(3) 如图 11-2 和图 11-3 所示，对于某些外部中断(如 TMR1IF、TMR2IF 和 TXIF)，可以使用除 GIE 位以外的 PEIE 标志位。

427

例 11-1 写出完成下面操作的指令：(a) 允许定时器 0 中断和外部硬件中断 0；(b) 禁用(屏蔽)定时器 0 中断；(c) 使用一条指令禁用(屏蔽)所有的中断。

解：

(a)
```
BSF  INTCON,TMR0IE   ;enable(unmask) Timer0 interrupt
BSF  INTCON,INT0IE ;enable external interrupt 1(INT0)
BSF  INTCON,GIE ;allow interrupts to come in
```

使用下面的两条指令可以完成上面的任务：

```
MOVLW B'10110000'    ;GIE = 1, TMR0IF = 1,INTIF0 = 1
MOVWF INTCON         ;load the INTCON reg
```

(b)
```
BCF  INTCON,TMR0IE ;mask (disable) Timer0 interrupt
```

```
(c) BCF  INTCON,GIE        ;mask all interrupts globally
```

11.1.7 复习题

1. 在中断和查询方法中，哪个方法可以避免束缚微控制器?
2. 请说出 INTCON 寄存器里的中断名称。
3. 当 PIC18 上电复位时，存储器的哪个区域会分配给中断向量表? 程序员可以修改向量表的存储器位置吗?
4. INTCON 寄存器的 D7(GIE)位在复位时的内容是什么，有什么含义?
5. 请给出允许 TMR0 中断的指令。
6. 在中断向量表中，哪些地址是分配给高优先级中断和低优先级中断的?

428

11.2 定时器中断编程

在第 9 章中已经讨论了如何使用查询方法来操作定时器 0、定时器 1、定时器 2 和定时器 3。本节将介绍如何使用中断方法对 PIC18 定时器进行编程。在学习本节之前，请复习第 9 章。

11.2.1 定时器复零标志位和中断

在第 9 章中已经提到定时器标志位将在定时器复零时变为高电平，还介绍了如何使用 BTFSS TMR0IF 指令监视定时器的标志位。在查询 TMR0IF 时，不得不等到 TMR0IF 变为高电平。这种方法的一个缺点是，微控制器在等待 TMR0IF 变为高电平的过程中被绑定而不能执行其他的任务。使用中断方法就可以避免微控制器的绑定。如果启用定时器的中断寄存器的中断位，那么在定时器复零和微控制器跳转至中断向量表执行 ISR 时，TMR0IF 就变为高电平。这样，微控制器在被告知定时器复零前就可以做其他的事情。要用中断方式代替查询方式，首先是必须允许中断，因为所有的中断在上电复位的时候都是被屏蔽的。TMRxIE 位用于使能给定的定时器的中断。TMRxIE 位包含在表 11-2 所列的不同寄存器里。对于定时器 0，INTCON 寄存器(如图 11-4 所示)包含 TMR0IE 位，而 PIE1(外围中断允许)则包含 TMR1IE 位。请参阅图 11-5 和程序 11-1。

表 11-2 定时器中断标志位和有关的寄存器

中断	标志位	寄存器	使能位	寄存器
Timer0	TMR0IF	INTCON	TMR0IE	INTCON
Timer1	TMR1IF	PIR1	TMR1IE	PIE1
Timer2	TMR2IF	PIR1	TMR2IE	PIE1
Timer3	TMR3IF	PIR3	TMR3IE	PIE2

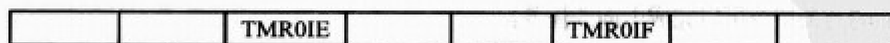


图 11-4 INTCON 寄存器的定时器 0 中断使能位和中断标志位

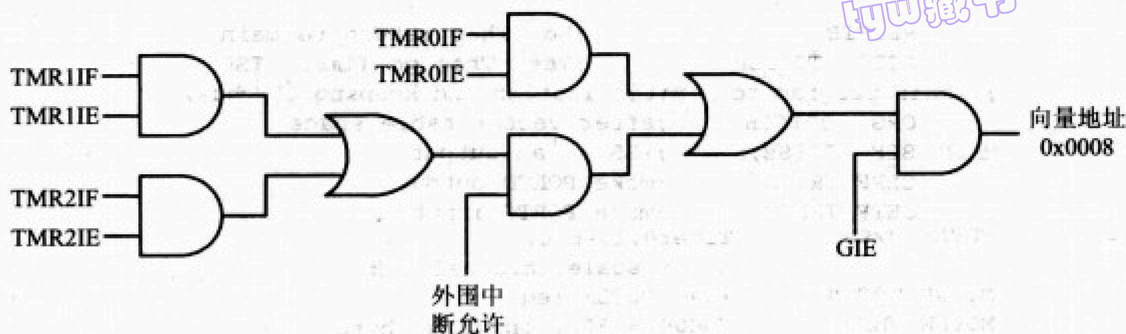


图 11-5 定时器中断使能标志位(TMRxIE)的作用

注意,图 11-5 中没有画出 TMRxIP(定时器中断优先)标志。TMRxIP 用于强行在 0x0018 向量处中断。详见 11.6 节。

429

关于程序 11-1,要注意以下几点。

(1) 必须避免使用中断向量表的存储器空间。因此,可以把所有的初始化代码放在诸如起始地址为 100H 的存储器空间。在上电复位期间,当 PIC18 在地址 0000 处被唤醒时执行的第一条指令就是 GOTO 指令。位于地址 0000 处的 GOTO 指令重定向控制器远离中断向量表。

(2) 在 MAIN 程序中,在使用 BSF INTCON, TMR0IE 指令允许(无掩蔽的)定时器 0 中断后,紧接着使用 BSF INTCON, GIE 指令允许全局中断。

(3) 在 MAIN 程序中,初始化定时器 0 寄存器,然后进入一个无限循环以保持 CPU 的忙状态。这就是 CPU 在执行很实际的应用。在这个例子里,循环从 PORTC 读取数据,然后传送到 PORTD。当数据不断从 PORTC 输入并输出到 PORTD 时,TMR0IF 标志位在定时器 0 复零时变为高电平,从而有微控制器跳出循环,跳转到地址 00008H 处执行对应于定时器 0 的 ISR。此时,PIC18 清除 GIE 位(INTCON 的 D7),以表明它正在执行中断,不能再被中断。换句话说,在中断里面不能再被中断。在 11.6 节将讨论在一个中断中允许另一个中断的情况。

(4) 定时器 0 的 ISR 放置在起始地址为 00200H 的存储地址,因为它太大以致于不能放入高优先级中断的地址空间 08～17H。

(5) 在定时器 0 的 ISR 里,注意 BCF INTCON, TMR0IF 指令要放在 RETFIE 指令前。这可以保证一个中断只被响应一次,而不会被当成多个中断。

(6) RETFIE 必须是 ISR 的最后一条指令。在执行 RETFIE 指令时,PIC18 自动允许 GIE 位(INTCON 寄存器的 D7 位),表明它已准备就绪受理新的中断。

程序 11-1:在这个程序里,假设 PORTC 和 8 个开关相连,PORTD 和 8 个 LED 相连。该程序使用定时器 0 在 PORTB.5 引脚产生一个方波,同时将数据从 PORTC 传送到 PORTD。

程序 11-1

```
;Program 11-1
        ORG   0000H
        GOTO  MAIN        ;bypass interrupt vector table
    ;--on default all interrupts land at address 00008
        ORG   0008H       ;interrupt vector table
        BTFSS INTCON,TMR0IF   ;Timer0 interrupt?
```

```
        RETFIE                  ;No. Then return to main
        GOTO   T0_ISR           ;Yes. Then go Timer0 ISR
;--main program for initialization and keeping CPU busy
        ORG    00100H      ;after vector table space
MAIN    BCF    TRISB,5     ;PB5 as an output
        CLRF TRISD         ;make PORTD output
        SETF TRISC         ;make PORTC input
    MOVLW 0x08         ;Timer0,16-bit,
                       ;no prescale,internal clk
    MOVWF T0CON        ;load T0CON reg
    MOVLW  0xFF        ;TMR0H = FFH, the high byte
    MOVWF  TMR0H       ;load Timer0 high byte
    MOVLW  0xF2        ;TMR0L = F2H, the low byte
    MOVWF TMR0L        ;load Timer0 low byte
    BCF INTCON,TMR0IF;clear timer interrupt flag bit
    BSF T0CON,TMR0ON        ;start Timer0
    BSF INTCON,TMR0IE       ;enable Timer 0 interrupt
    BSF INTCON,GIE     ;enable interrupts globally
;--keeping CPU busy waiting for interrupt
OVER  MOVFF PORTC,PORTD ;send data from PORTC to PORTD
    BRA OVER                ;stay in this loop forever
;---------------------------ISR for Timer 0
T0_ISR
    ORG 200H
    MOVLW  0xFF        ;TMR0H = FFH, the high byte
    MOVWF  TMR0H       ;load Timer0 high byte
    MOVLW  0xF2        ;TMR0L = F2H, the low byte
    MOVWF TMR0L        ;load Timer0 low byte
    BTG   PORTB,5      ;toggle RB5
    BCF INTCON,TMR0IF ;clear timer interrupt flag bit
EXIT  RETFIE ;return from interrupt (See Example 11-2)
    END
```

例 11-2 RETURN 和 RETFIE 指令的区别是什么？为什么不能在 ISR 中使用 RETURN 代替 RETFIE 作为最后的一条指令？

解：

两个指令都是执行将栈顶字节出栈到程序计数器，让 PIC18 返回到离开的地方。不过，RETFIE 还额外地执行 GIE 标志位的清零工作，表明中断已经执行完毕，PIC 现在可以接收新的中断。如果使用 RETURN 代替 RETFIE 作为中断服务程序的最后一条指令，就会在第一个中断后阻止其他新的中断，因为 GIE 表明中断仍在执行。

在程序 11-1 中，定时器 0 的 ISR(中断服务程序)太长，以致于不能将其放在高优先级中断的存储地址(0008～00017H)。然而，还是有足够的空间来查询地址 0008 上的中断类型。假设 PIC18 有很多的中断源，而地址 0008 只有有限的空间，这通常需要从地址 0008 跳转到有更大空间的地址来查询中断源。请参阅程序 11-2。

程序 11-2 使用定时器 0 和定时器 1 中断分别在引脚 RB1 和 RB7 上产生方波，同时将数据从 PORTC 传送到 PORTD。

程序 11-2

```
;Program 11-2
        ORG   0000H
        GOTO  MAIN         ;bypass interrupt vector table
    ;--on default all interrupts land at address 00008
        ORG   0008H        ;interrupt vector table
        GOTO  CHK_INT      ;go to an address with more space
    ;--check to see the source of interrupt
        ORG   0040H        ;we got here from 0008
    CHK_INT
        BTFSC INTCON,TMR0IF    ;Is it Timer0 interrupt?
        BRA   T0_ISR           ;Yes. Then branch to T0_ISR
        BTFSC PIR1,TMR1IF      ;Is it Timer1 interrupt?
        BRA   T1_ISR           ;Yes. Then branch to T1_ISR
        RETFIE                 ;No. Then return to main
    ;—main program for initialization and keeping CPU busy
        ORG   0100H  ;somewhere after vector table space
    MAIN BCF  TRISB,1      ;PB1 as an output
        BCF   TRISB,7      ;PB7 as an output
        CLRF  TRISD        ;make PORTD output
        SETF  TRISC        ;make PORTC input
        MOVLW 0x08         ;Timer0,16-bit,
                           ;no prescale,internal clk
        MOVWF T0CON        ;load T0CON reg
        MOVLW  0xFF        ;TMR0H = FFH, the high byte
        MOVWF  TMR0H       ;load Timer0 high byte
        MOVLW 0xF2         ;TMR0L = F2H, the low byte
        MOVWF TMR0L        ;load Timer0 low byte
        BCF INTCON,TMR0IF;clear Timer0 interrupt flag bit
        MOVLW 0x0          ;Timer1,16-bit,
                           ;no prescale,internal clk
        MOVWF T1CON        ;load T1CON reg
        MOVLW  0xFF        ;TMR1H = FFH, the high byte
        MOVWF  TMR1H       ;load Timer0 high byte
        MOVLW 0xF2         ;TMR1L = F2H, the low byte
        MOVWF TMR1L        ;load Timer1 low byte
        BCF PIR1,TMR1IF ;clear Timer1 interrupt flag bit
        BSF INTCON,TMR0IE      ;enable Timer0 interrupt
        BSF PIE1,TMR1IE        ;enable Timer1 interrupt
        BSF INTCON,PEIE ;enable peripheral interrupts
        BSF INTCON,GIE         ;enable interrupts globally
        BSF T0CON,TMR0ON       ;start Timer0
        BSF T1CON,TMR1ON       ;start Timer1
    ;--keeping CPU busy waiting for interrupt
    OVER MOVFF PORTC,PORTD ;send data from PORTC to PORTD
        BRA OVER               ;stay in this loop forever
    ;--------------------------ISR for Timer 0

    T0_ISR
        ORG 200H
        MOVLW  0xFF        ;TMR0H = FFH, the high byte
        MOVWF  TMR0H       ;load Timer0 high byte
        MOVLW 0xF2         ;TMR0L = F2H, the low byte
        MOVWF TMR0L        ;load Timer0 low byte
        BTG   PORTB,1      ;toggle PB1
        BCF INTCON,TMR0IF ;clear timer interrupt flag bit
```

```
        GOTO  CHK_INT
;-------------------------ISR for Timer1
T1_ISR
        ORG  300H
        MOVLW  0xFF        ;TMR1H = FFH, the high byte
        MOVWF  TMR1H       ;load Timer0 high byte
        MOVLW 0xF2         ;TMR1L = F2H, the low byte
        MOVWF  TMR1L       ;load Timer1 low byte
        BTG  PORTB,7
        BCF  PIR1,TMR1IF  ;clear Timer1 interrupt flag bit
        GOTO  CHK_INT
        END
```

注意,在程序 11-2 中用到的地址 0040H、0100H、00200H 和 0300H 都是随意的,也可以换成期望的任何地址。唯一不能选择的地址是上电复位地址 0000 和高优先级地址 0008,因为它们是在 PIC18 设计时就已经固定的。

程序 11-3 有两个中断:(1)PORTD 在定时器 0 每次溢出时计数;定时器 0 使用 16 位模式和最大的预分频器值;(2)当定时器 1 用作计数器时,将 1 Hz 脉冲送入定时器 1;当计数到 200 时,翻转引脚 PORTB.6。

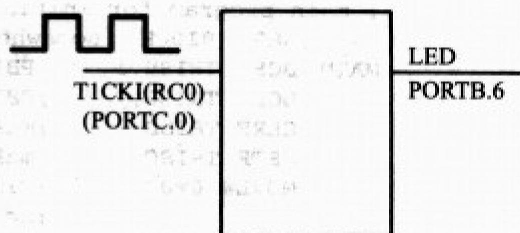


图 11-6　用于程序 11-3

程序　11-3

```
;Program 11-3
          ORG   0000H
          GOTO  MAIN         ;bypass interrupt vector table
      ;--on default all interrupts land at address 00008
          ORG   0008H
          GOTO  CHK_INT
;-----------------find the interrupt source
      ORG   0040H
CHK_INT
      BTFSC  INTCON,TMR0IF   ;Is it Timer0 interrupt?
      BRA   T0_ISR           ;Yes. Then branch to T0_ISR
      BTFSC  PIR1,TMR1IF     ;Is it Timer1 interrupt?
      BRA   T1_ISR           ;Yes. Then branch to T1_ISR
      RETFIE                 ;No. Then return to main
;--the main program for initialization
      ORG   00100H        ;after vector table space
MAIN  BSF   TRISC,T13CKI     ;PORTC.0 as an input
      CLRF  TRISD         ;make PORTD output
      BCF  TRISB,6        ;make RB6 output
      MOVLW  0x08         ;16-bit, prescale = 256,
                          ;internal clk
      MOVWF  T0CON        ;load T0CON reg
      MOVLW  0x00         ;TMR0H = 00H, the high byte
      MOVWF  TMR0H        ;load Timer0 high byte
      MOVLW 0x00          ;TMR0L = 0, the low byte
      MOVWF  TMR0L        ;load Timer0 low byte
```

```
        BCF INTCON,TMR0IF ;clear timer interrupt flag bit
        MOVLW 0x6         ;Timer1, no prescale,
                          ;ext. clock
        MOVWF T1CON       ;load T1CON reg
        MOVLW D'255'      ;TMR1H = 255
        MOVWF TMR1H       ;load Timer1 high byte
        MOVLW -D'200'     ;TMR1L = 0
        MOVWF TMR1L       ;load Timer1 low byte
        BCF PIR1,TMR1IF ;clear timer interrupt flag bit
        BSF T0CON,TMR0ON     ;start Timer0
        BSF T1CON,TMR1ON     ;start Timer1
        BSF INTCON,TMR0IE    ;enable Timer0 interrupt
        BSF PIE1,TMR1IE      ;enable Timer1 interrupt
        BSF INTCON,PEIE ;enable peripheral interrupts
        BSF INTCON,GIE  ;enable interrupts globally
OVER    BRA OVER        ;stay in this loop forever
;-------------------------ISR for Timer0
T0_ISR
        ORG 200H
        INCF PORTD      ;increment PORTD
        MOVLW 0x00      ;TMR0L = 0, the low byte
        MOVWF TMR0L     ;load Timer0 low byte
        MOVLW 0x00      ;TMR0H = 00, the high byte
        MOVWF TMR0H     ;load Timer0 high byte
        BCF INTCON,TMR0IF ;clear timer interrupt flag bit
        GOTO CHK_INT
;-------------------------ISR for Timer2
T1_ISR
        ORG 300H
        BTG PORTB,6     ;toggle PORTC.6
        MOVLW D'255'    ;TMR1H = 255
        MOVWF TMR1H     ;load Timer1 high byte
        MOVLW -D'200'   ;TMR1L = 0
        MOVWF TMR1L     ;load Timer1 low byte
        BCF PIR1,TMR1IF ;clear Timer1 interrupt flag bit
        GOTO CHK_INT
        END
```

注意，在程序11-2和程序11-3中，使用了GOTO CHK_INT指令代替RETFIE作为ISR的最后一条指令，这是因为要检查多个中断的活动。

434

11.2.2 使用C18编译器的PIC18中断C编程

在第7章中已经讨论了C18编译器如何使用伪指令#pragma code把代码放入到特定的ROM地址。因为C18不能自动地把ISR放入中断向量表，所以必须在中断向量表里使用汇编语言指令GOTO转移控制到ISR。其实现方法如下。

```
#pragma code high_vector =0x0008  // High-priority interrupt location
void My_HiVect_Int (void)
{
_asm
GOTO my_isr
```

```
_endasm
}
#pragma code                         // End of code
```

现在将微控制器从地址 00008 重定向到另一个程序来查找中断源，最终转移到 ISR。这需要使用关键字 **interrupt** 来完成，用法如下。

```
#pragma interrupt my_isr   //interrupt is reserved keyword
void my_isr (void)         //used for high-priority interrupt
{

//C18 places RETFIE here automatically due to
//interrupt keyword
}
```

注意，pragma、code 和 interrupt 是保留关键字，而其他的标号则可以自由选择。请参阅程序 11-2C 和程序 11-3C，它们分别是程序 11-2 和程序 11-3 的 C 版本。

程序 11-2C 使用定时器 0 和定时器 1 中断分别在引脚 RB1 和 RB7 上产生方波，同时将数据从 PORTC 传送到 PORTD。这是程序 11-2 的 C 版本。

程序　11-2C

```
//Program 11-2C (C version of Program 11-2)
#include <p18F458.h>
#define myPB1bit PORTBbits.RB1
#define myPB7bit PORTBbits.RB7

void T0_ISR(void);
void T1_ISR(void);

#pragma interrupt chk_isr  //used for high-priority
                           //interrupt only
void chk_isr (void)
{
      if (INTCONbits.TMR0IF==1)   //Timer0 causes interrupt?
           T0_ISR();              //Yes. Execute Timer0 ISR
      if(PIR1bits.TMR1IF==1)      //Or was it Timer1?
          T1_ISR();              // Yes. Execute Timer1 ISR
}

#pragma code My_HiPrio_Int=0x08//high-priority interrupt
void My_HiPrio_Int (void)
{
  _asm
    GOTO chk_isr
  _endasm
}
#pragma code
void main(void)
  {
    TRISBbits.TRISB1=0;     //RB1 = OUTPUT
    TRISBbits.TRISB7=0;     //RB7 = OUTPUT
    TRISC = 255;            //PORTC = INPUT
```

435

```
    TRISD = 0;          //PORTD = OUTPUT
    T0CON=0x0;          //Timer 0, 16-bit mode, no prescaler
    TMR0H=0x35;         //load TH0
    TMR0L=0x00;         //load TL0
    T1CON=0x88;         //Timer 1, 16-bit mode, no prescaler
    TMR1H=0x35;         //load TH1
    TMR1L=0x00;         //load TL1
    INTCONbits.TMR0IF=0;    //clear TF0
    PIR1bits.TMR1IF=0;      //clear TF1
    INTCONbits.TMR0IE=1;    //enable Timer0 interrupt
    INTCONbits.TMR0IE=1;    //enable Timer1 interrupt
    T0CONbits.TMR0ON=1;     //turn on Timer0
    T1CONbits.TMR1ON=1;     //turn on Timer1
    INTCONbits.PEIE=1;//enable all peripheral interrupts
    INTCONbits.GIE=1; //enable all interrupts globally
   while(1)      //keep looping until interrupt comes
    {
      PORTD=PORTC;    //send data from PORTC to PORTD
    }
  }

void T0_ISR(void)
  {
    myPB1bit=~myPB1bit;     //toggle PORTB.1
    TMR0H=0x35;             //load TH0
    TMR0L=0x00;             //load TL0
    INTCONbits.TMR0IF=0;    //clear TF0
  }

void T1_ISR(void)
  {
    myPB7bit=~myPB7bit;     //toggle PORTB.7
    TMR1H=0x35;             //load TH0
    TMR1L=0x00;             //load TL0
    PIR1bits.TMR1IF=0;      //clear TF1
  }
```

436

程序 11-3C 是程序 11-3 的C版本。

程序 11-3 有两个中断:(1)PORTD 在定时器 0 每次复零时计数;定时器 0 使用 16 位模式和最大的预分频器值;(2)当定时器 1 用作计数器时,将 1 Hz 脉冲送入到定时器 1;当计数到 200 时,翻转引脚 RB6。

程序 11-3C

```
//Program 11-3C
#include <p18F458.h>
#define myPB6bit PORTBbits.RB6

void chk_isr(void);
void T0_ISR(void);
void T1_ISR(void);
#pragma interrupt chk_isr //for high-priority interrupt only
void chk_isr (void)
```

```
{
if (INTCONbits.TMR0IF==1)  //Timer0 causes interrupt?
T0_ISR( );                 //Yes. Execute Timer0 program
     if(PIR1bits.TMR1IF==1)//Or was it Timer2?
     T1_ISR();             //Yes. Execute Timer2 program
     }

#pragma code My_HiPrio_Int=0x0008 //high-priority interrupt
void My_HiPrio_Int (void)
{
_asm
GOTO chk_isr
_endasm
}
#pragma code

void main(void)
  {
     TRISBbits.TRISB6=0;   //RB6 = OUTPUT
     TRISCbits.TRISC0=1;   //PORTC0 = INPUT
     TRISD=0;
     T0CON=0x08;           //Timer0, 16-bit mode,
                           //no prescaler
     TMR0H=0;              //load Timer0 high byte
     TMR0L=0;              //load Timer0 low byte
     T1CON=0x06;           //Timer 2, no prescaler
     TMR1H=255;            //load Timer1 high byte
     TMR1L=-200;           //load Timer1 low byte
     INTCONbits.TMR0IF=0;  //clear TF0
     PIR1bits.TMR1IF=0;    //clear TF1
     INTCONbits.TMR0IE=1;   //enable Timer0 interrupt
     PIE1bits.TMR1IE=1;     //enable Timer1 interrupt
     T0CONbits.TMR0ON=1;    //turn on Timer0
     T1CONbits.TMR1ON=1;    //turn on Timer1
     INTCONbits.PEIE=1;//enable all peripheral interrupts
     INTCONbits.GIE=1;      //enable all interrupts globally
     while(1);  //keep looping until interrupt comes
  }
void T0_ISR(void)
  {
    PORTD++;               //count up PORTD
    TMR0H=0;               //load Timer0 high byte
    TMR0L=0;               //load Timer0 low byte
    INTCONbits.TMR0IF=0;   //clear TF0
  }
void T1_ISR(void)
  {
    myPB6bit=~myPB6bit;    //toggle PB.6
    TMR1H=255;             //load Timer1 high byte
    TMR1L=-200;            //load Timer1 low byte
    PIR1bits.TMR2IF=0;     //clear TF1
  }
```

437

11.2.3 复习题

1. 判断对错：定时器0～定时器3在中断向量表里面分别分配有一个唯一的地址。
2. 在上电复位时，中断向量表的哪个地址会分配给高优先级向量？
3. TMR1IE属于哪个寄存器？如何使能该位？
4. 假设定时器1用于8位模式，TMR1L=F5H，TMR1IF位使能。解释定时器中断是如何工作的。
5. 判断对错：定时器0的ISR的最后两条指令是：

```
BCF   INTCON,TMR0IF
RETFIE
```

438

11.3 外部硬件中断编程

PIC18有3个外部硬件中断。引脚RB0(PORTB.0)、RB1(PORTB.1)和RB2(PORTB.2)分别被指定为INT0、INT1和INT2，用作外部硬件中断。当这些引脚被激活时，PIC18就会中断当前的任务，跳转到向量表执行中断服务程序。本节将通过一些汇编和C语言的例子来学习PIC18的这3个外部硬件中断。

11.3.1 外部中断INT0、INT1和INT2

PIC18有3个外部硬件中断：INT0、INT1和INT2，它们分别位于引脚RB0、RB1和RB2。如图11-7和图11-8所示。3个硬件中断在默认情况下全部指向向量表地址0008H，除非有特别的指定。在这些中断生效之前，必须使能它们，这由INTxIE位来完成。同INTxIE位有关的寄存器如表11-3所示。例如，使用指令 BSF INTCON,INT0IE 允许INT0。INT0是上升沿触发中断，也就是说，当一个由低变高的信号施加到引脚RB0(PORTB.0)时，INT0IF将会变为高电平，从而引起控制器的中断。INT0IF变为高电平让PIC18跳转到向量表地址0008H处执行中断。在表11-3中，要注意INTxIF位和它所在的寄存器。在上电复位时，PIC18将INT0、INT1和INT2设置为上升(正向)沿触发中断。为了使用下降(负向)沿触发中断，必须对INTEDGx位编程，这将在稍后学习到。

请研究程序11-4和它的C版本(程序11-4C)，以更好地了解外部硬件中断。

表11-3 硬件中断标志位和有关的寄存器

中断引脚	标志位	寄存器	使能位	寄存器
INT0(RB0)	INT0IF	INTCON	INT0IE	INTCON
INT1(RB1)	INT1IF	INTCON3	INT1IE	INTCON3
INT2(RB2)	INT2IF	INTCON3	INT2IE	INTCON3

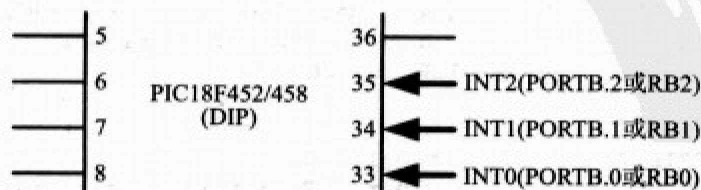


图11-7 PIC18外部硬件中断引脚

439

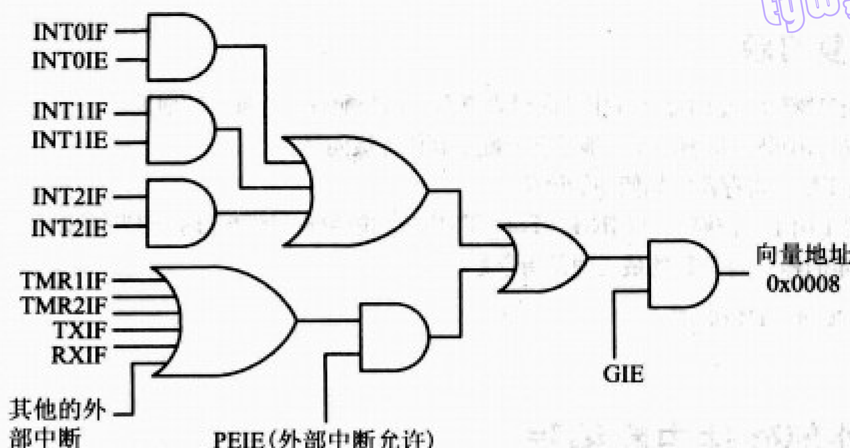


图 11-8　INT0～INT2 硬件中断

程序 11-4 连接一个开关到 INT0，并连接一个 LED 到引脚 RB7。在这个程序中，每次 INT0 被激活时，它就翻转 LED，同时将数据从 PORTC 传送到 PORTD。

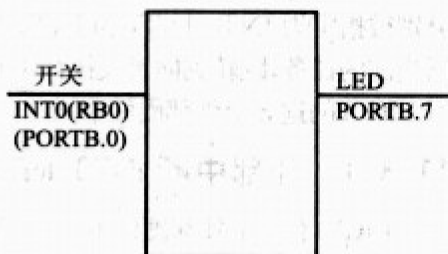


图 11-9　用于程序 11-4

程序　11-4

```
;Program 11-4
       ORG   0000H
       GOTO  MAIN                ;bypass interrupt vector table
;--on default all interrupts go to to address 00008
       ORG   0008H               ;interrupt vector table
       BTFSS INTCON,INT0IF       ;Did we get here due to INT0?
       RETFIE                    ;No. Then return to main
       GOTO  INT0_ISR            ;Yes. Then go INT0 ISR
;;--the main program for initialization
       ORG   00100H
MAIN BCF   TRISB,7               ;PB7 as an output
       BSF TRISB,INT0            ;make INT0 an input pin
       CLRF TRISD                ;make PORTD output
       SETF TRISC                ;make PORTC input
       BSF INTCON,INT0IE         ;enable INT0 interrupt
       BSF INTCON,GIE            ;enable interrupts globally
OVER   MOVFF PORTC,PORTD         ;send data from PORTC to PORTD
       BRA OVER                  ;stay in this loop forever
;--------------------------ISR for INT0
INT0_ISR
       ORG 200H
       BTG PORTB,7               ;toggle PB7
       BCF INTCON,INT0IF         ;clear INT0 interrupt flag bit
       RETFIE                    ;return from ISR
       END
```

观察程序 11-4。当一个上升沿信号施加到引脚 INT0 时，LED 就会被翻转。在这个例子中，要再次翻转 LED，则 INT0 脉冲必须变回到低电平，然后再变为高电平来产生上升沿以引起中断。

程序 11-4C

```
//Program 11-4C (This is the C version of Program 11-4)
      #include <p18F4580.h>
      #define mybit PORTBbits.RB7
      void chk_isr(void);
      void INT0_ISR(void);
      #pragma interrupt chk_isr//used for high-priority int
      void chk_isr (void)
      {
      if (INTCONbits.INT0IF==1)   //INT0 caused interrupt?
      INT0_ISR( );                //Yes. Execute INT0 program
      }
      #pragma code My_HiPrio_Int=0x08  //high-priority
                                  //interrupt location
      void My_HiPrio_Int (void)
      {
      _asm
      GOTO chk_isr
      _endasm
      }
      #pragma code
      void main(void)
        {
            TRISBbits.TRISB7=0;    //RB7 = OUTPUT
            TRISBbits.TRISB0=1;    //INT0 = INPUT
            TRISC = 0xFF;          //PORTC = INPUT
            TRISD = 0;             //PORTD = OUTPUT
            INTCONbits.INT0IF=0;   //clear TF1
            INTCONbits.INT0IE=1;   //enable Timer0 interrupt
            INTCONbits.GIE=1;      //enable all interrupts
    while(1)      //keep looping until interrupt comes
      {
       PORTD=PORTC;
      }
    }
void INT0_ISR(void)
   {
     mybit=~mybit;
     INTCONbits.INT0IF=0;          //clear INT0 flag
   }
```


11.3.2 下降沿触发中断

在上电复位时，PIC18 自动地将 INT0、INT1 和 INT2 设为正向(上升)沿触发中断。要使它们变成负向(下降)沿触发中断，则必须对对应的 INTEDGx 位编程，其中 x 可以是 0、1 或者 2。INTCON2 寄存器包含有 INTEDG0、INTEDG1 和 INTEDG2 标志位，如图 11-10 所示。INTEDG0、INTEDG1 和 INTEDG2 分别是 INTCON2 的 D4 位、D5 位和 D6 位，如图 11-10 所示。这些位的状态决定了硬件中断是上升沿还是下降沿触发模式。在上电复位时，INT-

EDGx 位全部为 1，也就是说外部硬件中断全部是上升沿触发的。当把 INTEDG0 位被设为低电平时，INT0 的外部硬件中断就变成下降沿触发中断。例如，使用指令 BSF INTCON2, INTEDG1 把 INTEDG1 变成下降沿触发中断，也就是说，当一个由高变低的信号施加到引脚 RB1(PORTB.1)时，控制器就会中断，跳转到向量表地址 0008H 执行 ISR（假设 GIE 位和 INT0IE 位都被使能）；如程序 11-5 所示。程序 11-5C 是它的 C 版本。

	INTEDG0	INTEDG1	INTEDG2				

INTEDGx　外部硬件中断边沿触发位
0＝负向（下降）沿触发
1＝正向（上升）沿触发（上电复位时的默认值）

图 11-10　INTCON2 寄存器的 INTEDG 允许上升沿或者下降沿触发

在程序 11-5 中，假设引脚 RB1(INT1)连接到一个脉冲发生器，引脚 RB7 连接到一个 LED。程序在脉冲的下降沿翻转 LED。换句话说，LED 的开关切换速度与施加到 INT1 引脚的脉冲速度相同。

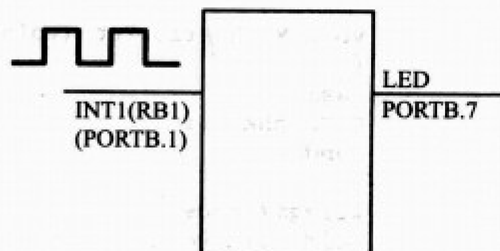


图 11-11　程序 11-5 示意图

程序　11-5

```
;Program 11-5
        ORG   0000H
        GOTO  MAIN          ;bypass interrupt vector table
    ;--on default all interrupts go to to address 00008
        ORG   0008H         ;interrupt vector table
        BTFSS INTCON3,INT1IF  ;Did we get here due to
                              ;INT1 interrupt?
        RETFIE                ;No. Then return to main
        GOTO   INT1_ISR       ;Yes. Then go INT1 ISR
    ;--the main program for initialization
        ORG   00100H
    MAIN BCF   TRISB,7      ;PB7 as an output
        BSF   TRISB,INT1   ;make INT1 an input pin
        BSF   INTCON3,INT1IE  ;enable INT1 interrupt
        BCF  INTCON2,INTEDG1  ;make it negative
                              ;edge-triggered
        BSF  INTCON,GIE       ;enable interrupts globally
    OVER BRA OVER             ;stay in this loop forever
    ;-------------------------ISR for INT1
    INT1_ISR
        ORG 200H
        BTG PORTB,7           ;toggle on RB7
        BCF INTCON3,INT1IF ;clear INT1 interrupt flag bit
        RETFIE
        END
```

442

程序 11-5C

```
//Program 11-5C (This is the C version of Program 11-5)
      #include <p18F4580.h>
      #define mybit PORTBbits.RB7
      void chk_isr(void);
      void INT1_ISR(void);
      #pragma code My_HiPrio_Int =0x0008    //high-priority
interrupt location
      void My_HiPrio_Int (void)
      {
      _asm
      GOTO chk_isr
      _endasm
      }
      #pragma code
      #pragma interrupt chk_isr  //used for high-priority
                                 //interrupt only
      void chk_isr (void)
      {
      if (INTCON3bits.INT1IF==1) //INT1 causes interrupt?
      INT1_ISR( );     //Yes. Execute INT1 program
            }
      void main(void)
        {
           TRISBbits.TRISB7=0;    //RB7 = OUTPUT
           TRISBbits.TRISB1=1;    //INT1 = INPUT
           INTCON3bits.INT1IF=0; //clear INT1
           INTCON3bits.INT1IE=1; //enable INT1 interrupt
           INTCON2bits.INTEDG1=0;//make it negative edge
           INTCONbits.GIE=1;     //enable all interrupts
           while(1);//keep looping until interrupt comes
  }

void INT1_ISR(void)
  {
    mybit=~mybit;
    INTCON3bits.INT1IF=0;        //clear INT1 flag
  }
```

443

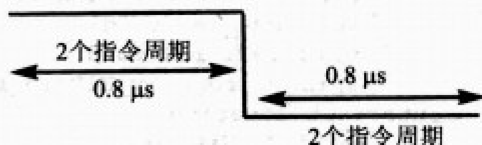
11.3.3 边沿触发中断采样

在本节结束之前,需要弄清楚边沿触发中断多久采样一次。对于边沿触发中断,外部中断源必须保持至少两个指令周期的高电位,然后同样保持至少两个指令周期的低电位,以保证微控制器能查询到电平的改变。

上升沿(或者是下降沿)被 PIC18 锁存,并包含在 INTxIF 位中。INT0IF、INT1IF 和 INT2IF 位包含着引脚 RB0~RB2 上的锁存的上升沿(或者是下降沿,这由 INTEDGx 位决定)。INT0IF~INT2IF 位的功能是中断服务标志位。当一个中断服务标志位为高电平时,它向外界表明正在执行中断服务,在该中断服务执行完毕之前,微控制器将不应答来自 INTn 引脚上任何新的中断请求。这就好比你呼叫一个正在通话的号码时听到的忙音。对于 INT0IF~INT2IF,还要注意一点的是,在 ISR 完成之前(也就是执行 RETFIE 指令前),必须将这些

位(INT0IF~INT2IF)清零,以表明中断已经完成,PIC18 已经准备就绪响应来自该引脚上的另一个中断。要识别一个新的中断,必须将该引脚返回到逻辑低电平状态,然后再回到高电平来作为上升沿触发中断。

探测边沿触发的最小脉冲持续时间=2 指令周期
对于 XTAL=10 MHz,可以得到一个指令周期时间是 400 ns=0.4 μs



11.3.4 复习题

1. 判断对错:在上电复位时,全部外部硬件中断 INT0~INT2 都跳转到中断向量表地址 0008。
2. 对于 PIC18F458,哪些引脚是分配给 INT0~INT2 的?
3. 说明如何使能 INT1。
4. 假设外部硬件中断 INT0 的 INT0IE 位有效。解释当它被激活时是如何工作的。
5. 判断对错:在上电复位时,外部硬件中断是下降沿触发的。
6. 如何判断单个中断不会被识别成多个中断呢?
7. 判断对错:INT0 的 ISR 最后两条指令是:

444

```
BCF   INTCON2,INT0IF
RETFIE
```

11.4 串行通信中断编程

在第 10 章中已经学习了 PIC18 的串行通信,在那里全部的例子都使用的是查询方法。本节将研究基于中断的串行通信。除了串行通信端口的数据发送和接收之外,PIC18 还能做更多的事情。

11.4.1 RCIF 和 TXIF 标志位与中断

回顾第 10 章所述,当帧数据的最后一位(即结束位)发送时,TXIF(发送中断)会变为高电平,表明 TXREG 寄存器已经准备好发送下一字节数据。当接收完整帧的数据(包括结束位)时,RCIF(接收中断)会变为高电平。换句话说,当 RCREG 寄存器里有一个字节数据时,RCIF 就会变为高电平,这表明接收到的字节需要在丢弃(被新字节覆盖)之前被取走。就串行通信而言,上面所有的概念都适用于查询方式和中断方式;唯一的区别是串行通信的工作方式。在查询方法中,需要等待标志位(TXIF 或 RCIF)变为高电平;而在等待期间微控制器不能做其他的事情。在中断方法中,当 PIC18 接收到一个字节或者准备就绪发送下一字节时,将会告诉微控制器。这样,微控制器在串行通信时也可以做其他的事情。

PIC18 有两个中断用于串行通信。一个中断用于发送,而另一个中断用于接收。如果 TXIE 或者 RCIE 的相应的中断位有效,那么当 TXIF 或者 RCIF 变为高电平时,PIC18 就会中断,并跳转到存储地址 0008H 处执行 ISR。

表 11-4 串行端口中断标志位和有关的寄存器

中断	标志位	寄存器	使能位	寄存器
TXIF(发送)	TXIF	PIR1	TXIE	PIE1
RCIF(接收)	RCIF	PIR1	RCIE	PIE1



图 11-12 PIE1 寄存器含有 TXIE 和 RCIE 位

445

11.4.2 使用 PIC18 中的串行 COM

在众多的应用中，串行中断主要用于数据接收，而很少用于数据串行发送。这类似于电话接听，需要响铃通知有电话进入。如果需要打一个电话，还有其他提醒的方法而不需要响铃。然而，在接听电话的时候，无论我们正在做什么工作，都必须立即做出回答，否则就会错过这个电话呼叫。相似地，使用串行中断来接收外来的数据就不会丢失数据。请参阅程序 11-6。注意 ISR 的最后一条指令是 RETFIE。由于向 TXREG 寄存器写入了字节数，所以无需执行对 TXIF 标志位的清零操作。

在图 11-13 中，注意 PEIE(外围中断使能)的功能是如何使能串行通信中断和其他中断的进入。这是对 11.1 节中关于 GIE 位的补充。

在程序 11-6 中，假设一个 8 位开关连接到 PORTD。在这个程序中，PIC18 从 PORTD 读取数据，然后不断地写入 TXREG 寄存器，用于串行传输。假设 XTAL=10 MHz。波特率设为 9600。

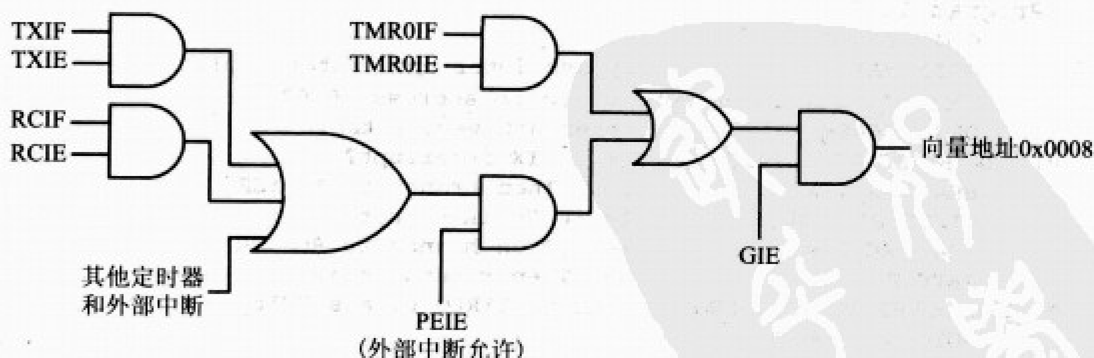


图 11-13 串行中断使能位

程序 11-6

```
;Program 11-6
        ORG   0000H
        GOTO  MAIN          ;bypass interrupt vector table
    ;--on default all interrupts go to to address 00008
        ORG   0008H         ;interrupt vector table
        BTFSC PIR1,TXIF     ;Is interrupt due to transmit?
        BRA TX_ISR          ;Yes. Then go to ISR
        RETFIE              ;No. Then return
        ORG 0040H
```

```
TX_ISR                  ;service routine for TXIF
      MOVWFF PORTD,TXREG;load new value, clear TXIF
      RETFIE            ;then return to main
;--the main program for initialization
      ORG   00100H
MAIN  SETF TRISD        ;make PORTD input
      MOVLW 0x20        ;enable transmit and choose
                        ;low baud
      MOVWF TXSTA       ;write to reg
      MOVLW D'15'       ;9600 bps
                        ;(Fosc / (64 * Speed) - 1)
      MOVWF SPBRG       ;write to reg
      BCF TRISC, TX     ;make TX pin of PORTC an
                        ;output pin
      BSF RCSTA, SPEN   ;enable the serial port
      BSF PIE1,TXIE     ;enable TX interrupt
      BSF INTCON,PEIE   ;enable peripheral interrupts
      BSF INTCON,GIE    ;enable interrupts globally
OVER  BRA OVER          ;stay in this loop forever
      END
```

446

程序 11-7 对程序 11-6 进行了修改,带有接收中断。在这个程序中,PIC18 从 PORTD 得到数据,然后不断地传送至 TXREG 寄存器,同时来自串行端口的数据也被送到 PORTB。假设 XTAL=10 MHz,波特率=9600。读者可以自己验证这个程序,把你的 PICTrainer 连接到 x86 IBM PC,再使用 HyperTerminal 程序在 PIC Trainer 和 IBM PC 之间发送和接收数据。

程序 11-7

```
;Program 11-7
      ORG   0000H
      GOTO  MAIN                ;bypass interrupt vector table
;--on default all interrupts go to to address 00008
      ORG   0008H               ;interrupt vector table
HI_ISR BTFSC PIR1,TXIF          ;is it TX interrupt?
      BRA   TX_ISR              ;Yes. Then branch to TX_ISR
      BTFSC PIR1,RCIF           ;Is it RC interrupt?
      BRA   RC_ISR              ;Yes. Then branch to RC_ISR
      RETFIE                    ;No. Then return to main
TX_ISR MOVFF PORTD,TXREG        ;loading TXREG clears TXIF
      GOTO HI_ISR
RC_ISR
      MOVFF RCREG,PORTB         ;copy received data to PORTB
      GOTO HI_ISR
;--the main program for initialization
      ORG   00100H
MAIN  CLRF TRISB          ;PORTB as an output
      SETF TRISD          ;make PORTD input
      MOVLW 0x20          ;enable transmit and choose low baud
      MOVWF TXSTA         ;write to reg
      MOVLW D'15'         ;9600 bps (Fosc / (64 * Speed) - 1)
      MOVWF SPBRG         ;write to reg
      BCF TRISC,TX        ;make TX pin of PORTC an output pin
      BSF TRISC,RX        ;make RCV pin of PORTC an input pin
```

```
        MOVLW 0x90        ;enable receive and serial port
        MOVWF RCSTA       ;write to reg
        BSF PIE1,TXIE     ;enable TX interrupt
        BSF PIE1,RCIE     ;enable receive interrupt
        BSF INTCON,PEIE   ;enable peripheral interrupts
        BSF INTCON,GIE    ;enable interrupts globally
OVER    BRA OVER          ;stay in this loop forever
        END
```

程序 11-7C

```
//Program 11-7C (This is the C version of Program 11-7)
#include <p18F458.h>
void chk_isr(void);
void TX_ISR(void);
void RC_ISR(void);
#pragma code My_HiPrio_Int=0x08 //high-priority interrupt
void My_HiPrio_Int (void)
{
     _asm
          GOTO chk_isr
     _endasm
}
#pragma code
#pragma interrupt chk_isr//used for high-priority interrupt
void chk_isr (void)
{
if (PIR1bits.TXIF==1)         //Transmit caused interrupt?
  TX_ISR( );                  //Yes. Execute Transmit program
if (PIR1bits.RCIF==1)         //Receive caused interrupt?
  RC_ISR( );                  //Yes. Execute Receive program
}
void main(void)
  {
     TRISD = 0xFF;            //PORTD = INPUT
     TRISB = 0;               //PORTB = OUTPUT
     TRISCbits.TRISC6=0;      //TX pin = OUTPUT
     TRISCbits.TRISC7=1;      //RCV pin = INPUT
     TXSTA=0x20;              //choose low baud rate, 8-bit
     SPBRG=15;                //9600 baud rate/ XTAL = 10 MHz
     RCSTAbits.CREN=1;
     RCSTAbits.SPEN=1;
     TXSTAbits.TXEN=1;
     PIE1bits.RCIE=1;         //enable RCV interrupt
     PIE1bits.TXIE=1;         //enable TX interrupt
     INTCONbits.PEIE=1;       //enable peripheral interrupts
     INTCONbits.GIE=1;        //enable all interrupts globally
     while(1);         //keep looping until interrupt comes
  }
void TX_ISR(void)
  {
     TXREG=PORTD;
  }
void RC_ISR(void)
```

447

```
    {
        PORTB=RCREG;
    }
}
```

11.4.3 复习题

1. 判断对错:包括TXIF和RXIF在内的所有中断都指向中断向量表中的同一个位置。
2. 中断向量表中的哪个地址是分配给串行中断的?
3. TXIF和RXIF属于哪个寄存器?如何使能该位呢?
4. 假设RCIF位使能。解释中断是如何被激活的,以及激活后的活动。
5. 判断对错:在上电复位时,串行中断被激活而且准备就绪。
6. 判断对错:接收中断ISR的最后两条指令是:

```
BCF  PIR1, RCIF
RETFIE
```

448

7. 如果是发送中断,请重新回答第6题。

11.5 PORTB变化中断

当查询到PORTB的4个引脚(RB4~RB7)的其中一个引脚有任何改变时,就会引起中断。将它们叫作"PORTB变化中断",以区别于同样来自PORTB(RB0~RB2)的INT0~INT2中断。如图11-15所示。PORTB变化中断有一个叫作RBIF的单独中断标志位,位于INTCON寄存器。如图11-14所示。在图11-14中,还要注意RBIE位是用于使能PORTB变化中断的。在11.3节中已讨论了外部硬件中断INT0、INT1和INT2。注意下面的PORTB变化中断和INT0~INT2中断之间的区别。

D7							D0
GIE				RBIE			RBIF

GIE(全局中断使能)
GIE=0 禁止所有的中断。如果GIE=0,不响应任何中断,即使它们各自的中断位是允许的。
如果GIE=1,允许中断发生。每个中断源在相应的中断使能位为1时有效

RBIE PORTB-变化中断使能
=0 禁止 PORTB变化中断
=1 允许 PORTB变化中断

RBIF PORTB-变化中断标志位
=0 RB4~RB7引脚都没有改变状态
=1 RB4~RB7引脚至少有一个改变状态

对于RB4~RB7引脚的任何状态改变要引起中断,RBIE位和GIE位都必须为高电平RB4~RB7引脚也必须配置为输入引脚,用于中断。为了对RBIF标志位清零,必须使用指令BCF INTCON, RBIF读取RB4~RB7引脚

图11-14 INTCON(中断控制)寄存器

(1) INT0～INT2 的每个中断都有自己的引脚而且相互独立。这些中断分别使用引脚 PORTB.0(RB0)、PORTB.1(RB1)和 PORTB.2(RB2)。而 PORTB 变化中断使用 PORTB 的 RB4～RB7 全部 4 个引脚。尽管使用的是 4 个引脚，可是仍只被看作一个中断。

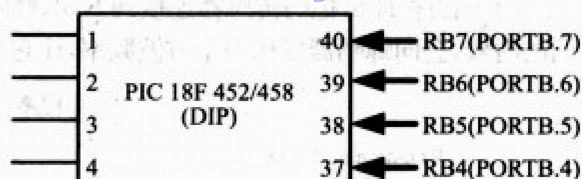


图 11-15　PORTB-变化中断引脚

(2) INT0～INT2 的每个中断都有自己的标志位，且相互独立，而 PORTB 变化中断只有一个标志位。

(3) INT0～INT2 的每个中断都可以用编程的方法来选择是下降沿还是上升沿触发，而 PORTB 变化中断是由其中一个引脚的状态变化来触发的，可以是从高电平到低电平的变化，也可以是从低电平到高电平的变化。如图 11-16 所示。

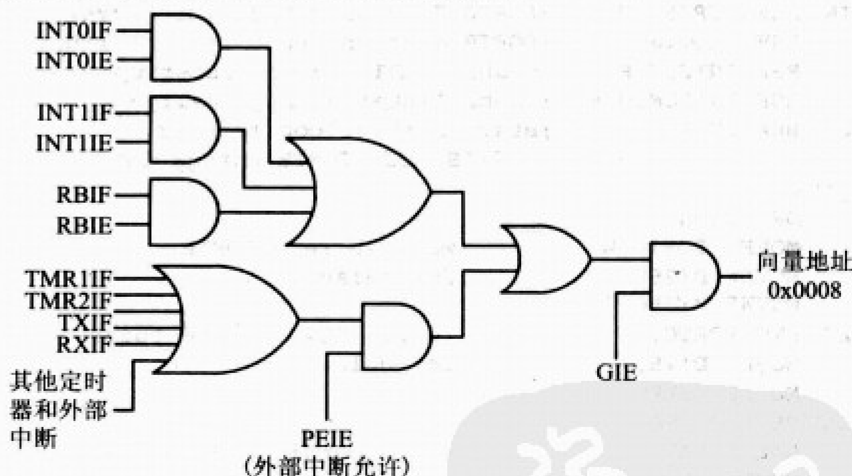


图 11-16　PORTB-变化中断(RBIF)

PORTB 变化中断常用于键盘接口，这将在第 12 章看到。另一个使用 PORTB 变化中断的方法请参阅程序 11-8。在这个程序中，假设一个门传感器连接到引脚 RB4，而每次开门或者关门的时候，蜂鸣器就会发声。如图 11-17 所示。

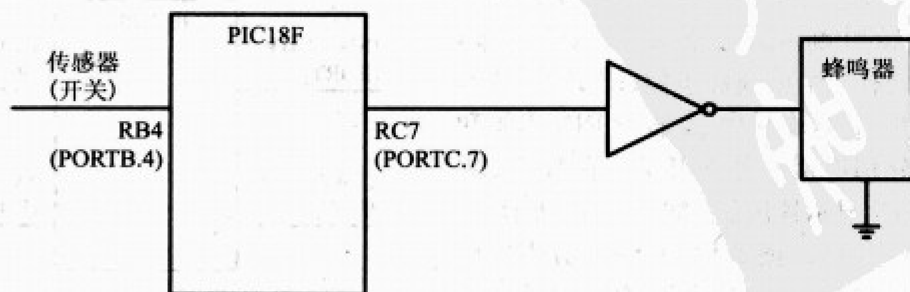


图 11-17　程序 11-8 的 PORTB-变化中断

对于程序11-8,把门传感器连接到RB4,蜂鸣器连接到引脚RC7。在这个程序中,每次开门的时候,会向蜂鸣器发送一个方波频率,让它发声。

程序 11-8

```
;Program 11-8
MYREG EQU   0x20         ;set aside a couple of registers
DELRG EQU 0x80           ;for buzzer time delay
      ORG   0000H
      GOTO  MAIN         ;bypass interrupt vector table
;--on default all interrupts go to to address 00008
      ORG   0008H              ;interrupt vector table
      BTFSS INTCON,RBIF        ;Did we get here due to RBIF?
      RETFIE                   ;No. Then return to main
      GOTO  PB_ISR             ;Yes. Then go ISR
;--the main program for initialization
      ORG   00100H
MAIN  BCF   TRISC,7      ;PORTC.7 as an output for buzzer
      BSF   TRISB,4      ;PORTB.4 as an input for interrupt
      BSF INTCON,RBIE    ;enable PORTB-Change interrupt
      BSF INTCON,GIE     ;enable interrupts globally
OVER  BRA OVER           ;stay in this loop forever
;------------------------ISR for PORTB-Change INT
PB_ISR
      ORG 200H
      MOVF   PORTB,W           ;we must read PORTB
      MOVLW D'250'             ;for delay
      MOVWF MYREG
BUZZ  BTG PORTC,7              ;toggle PC7 for the buzzer
      MOVLW D'255'             ;for delay
      MOVWF DELRG
DELAY DECF DELRG,F
      BNZ DELAY                ;keep sounding the buzzer
      DECF MYREG,F
      BNZ BUZZ
      BCF INTCON,RBIF ;and clear RBIF interrupt flag bit
      RETFIE
      END
```

注意,对于PORTB-变化中断,不需要使能PEIE;但是需要使能GIE位。

要再次注意的是,INT0～INT2中每个中断都有自己的中断标志位,而RB4～RB7全部4个引脚却只有一个中断标志位(RBIF)。请参阅程序11-9。在这个程序中,假设RB4引脚和RB5引脚分别连接有外部开关。在激活SW时,在LED上显示其状态。如图11-18所示。

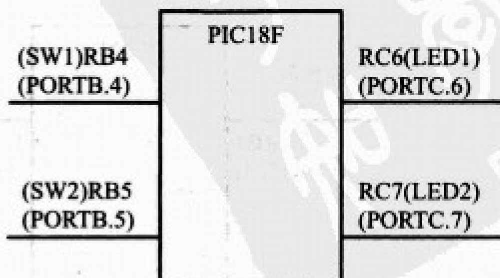


图11-18 程序11-9的PORTB-变化中断

451

对于程序11-9,把SW1和SW2分别连接到引脚RB4和RB5。在这个程序中,激活SW1和SW2将会分别改变LED1和LED2的状态。

程序 11-9

```
;Program 11-9
          ORG   0000H
          GOTO  MAIN  ;bypass interrupt vector table
    ;--on default all interrupts go to to address 00008
          ORG   0008H        ;interrupt vector table
          BTFSS INTCON,RBIF;Did we get here due to RBIF?
          RETFIE             ;No. Then return to main
          GOTO   PB_ISR      ;Yes. Then go ISR
    ;--the main program for initialization
          ORG   0100H
    MAIN  BCF TRISC,4        ;PC4 as an output
          BCF TRISC,5        ;PC5 as an output
          BSF TRISB,4 ;PB4 as an input for the interrupt
          BSF TRISB,5 ;PB5 as an input for the interrupt
          BSF INTCON,RBIE ;enable PORTB interrupt
          BSF INTCON,GIE  ;enable interrupts globally
    OVER  BRA OVER        ;stay in this loop forever
    ;-------------------------ISR for PORTB_Change
    PB_ISR
          ORG 200H
          MOVFF PORTB,W ;get the status of switches
          ANDLW 0x30 ;mask unneeded bits
          MOVFF W,PORTC ;update LEDs
          BCF INTCON,RBIF ;clear RBIF interrupt flag bit
          RETFIE
          END
```

452

程序 11-9C

```
//Program 11-9C (This is the C version of Program 11-9)
#include <p18F458.h>
#define LED1 PORTCbits.RC4
#define LED2 PORTCbits.RC5

void chk_isr(void);
void RBINT_ISR(void);

#pragma code My_HiPrio_Int =0x0008 //high-priority int
void My_HiPrio_Int (void)
{
     _asm
          GOTO chk_isr
    _endasm
}
#pragma code
#pragma interrupt chk_isr  //used for high-priority int
void chk_isr (void)
{
     if (INTCONbits.RBIF==1)    //RBIF caused interrupt?
     RBINT_ISR( );          //Yes. Execute ISR program
}
void main(void)
  {
     TRISCbits.TRISC4=0;   //RC4 = OUTPUT
     TRISCbits.TRISC5=0;   //RC5 = OUTPUT
```

```
        TRISBbits.TRISB4 = 1; //RB4 = INPUT for interrupt
        TRISBbits.TRISB5 = 1; //RB5 = INPUT for interrupt
        INTCONbits.RBIF=0;    //clear RBIF
        INTCONbits.RBIE=1;    //enable RB interrupt
        INTCONbits.GIE=1;     //enable all interrupts globally
        while(1);  //keep looping until interrupt comes
    }
  void RBINT_ISR(void)
    {
        LED1=PORTBbits.RB4;
        LED2=PORTBbits.RB5;
        INTCONbits.RBIF=0;    //clear RBIF flag
    }
```

复习题

1. 判断对错:每个 PORTB 引脚对应一个中断。
2. 分配给 PORTB 变化中断的中断向量表地址是什么?
3. RBIF 和 RBIE 标志位属于哪个寄存器? 如何使能这些标志位呢?
4. 请写出 PORTB 变化中断的 ISR 的最后两条指令。
5. 判断对错:在上电复位时,RBIF 中断被激活,而且准备就绪。

453

11.6 PIC18 的中断优先级

下面必须要谈到的主题是,如果两个中断同时被激活了,那将发生什么情况呢? 这两个中断的哪个会先被响应呢? 这就是本节要讨论的主题——中断优先级。

11.6.1 设置中断优先级

在 PIC18 微控制器中,有两个中断优先级:低级和高级。地址 0008 是分配给高优先级的中断,而低优先级的中断就指向中断向量表中的地址 00018。如表 11-5 所示。在上电复位期间,所有的中断都被配置为高优先级,并指向地址 0008。这样做的目的,是为了 PIC18 能与早期的 PIC 微控制器(如 PIC16xxx)兼容。通过对 RCON 寄存器的 IPEN(中断优先级使能)位编程,可以把 PIC18 配置成一个有两级优先级的系统。图 11-19给出了 RCON 寄存器的 IPEN 位。在上电复位时,IPEN 位为 0,因此 PIC18 是单优先级的芯片,这和 PIC16xxx 一样。为了将 PIC18 配置成两级优先级的系统,首先必须把 IPEN 位置为高电平。只有在 IPEN=1 以后,才能通过编程 IP(中断优先级)位来向任何中断分配低优先级。图 11-20 画出了带有 IP 位(即 TXIP、RCIP、TMR1IP 和 TMR2IF)的 IPR1(中断优先级寄存器)。如果 IPEN=1,那么 IP 位有效,可以对给定的中断分配低优先级。当一个中断被分配为低优先级后,它将指向中断向量表地址 0018 而不是 0008。IP(中断优先级)位和 IF(中断标志)位,IE(中断使能)位涵盖了 PIC18 编程的所有中断标志位。表 11-6 描述了 3 个标志位和在本章中用到的一些中断对应的寄存器。在表 11-6 中,注意没有 INT0 优先级标志位。INT0 只有一个优先级,也就是高优先级。换而言之,除外部硬件中断 INT0 外,PIC18 的所有中断都可以分配为高优先级或者低优先级。仔细研究图 11-22~图 11-25。在观察这些图的时候,要注意一点:当置 IPEN=1

时，就是允许中断优先级功能。现在可以通过设置两个位来允许中断：(a)必须置 GIEH=1，GIEH 位是 INTCON 寄存器的一部分(如图 11-21 所示)，这和前面已介绍的 GIE 位相似；(b)需要置 1 的第二位是 GIEL(INTCON 的一部分)。当 GIEL=1 时，就是允许所有 IP=0 的中断。因此，所有分配有低优先级的中断都指向向量地址 00018H。

表 11-5 PIC18 的中断向量表

中断	ROM 地址(十六进制)
上电复位	0000
高优先级中断	0008(上电复位时的默认值)
低优先级中断	0018(由 IP 位选择)

IPEN							

IPEN 中断优先级使能位
0=所有中断都指向向量表地址0008（默认）
1=中断可以分低或高优先级。

IPEN的重要性： 在上电复位时，PIC18的所有中断都指向地址0008，因此PIC18是单优先级系统，这和PIC16一样。为了把PIC18的中断分为低优先级或者高优先级，必须置IPEN=1。当IPEN=1时，可以通过改变中断的IPR(中断优先级寄存器)的相应位来为中断分配高优先级或者低优先级。当中断优先级使能(IPEN=1)时，必须把GIEH和GIEL位置为高电平，以使能全局中断。注意，在图11-21中GIE和GIEH是相同的

图 11-19 RCON 寄存器(IPEN 允许把中断分成两个优先级)

		RCIP	TXIP			TMR2IP	TMR1IP

RCIP USART（串行COM）接收中断优先级位
0=低优先级
1=高优先级

TXIP USART(串行COM)发送中断优先级位
0=低优先级
1=高优先级

TMR2IP 定时器2中断优先级位
0=低优先级
1=高优先级

TMR1IP 定时器1中断优先级位
0=低优先级
1=高优先级

图 11-20 IPR1 外部中断优先级寄存器 1

表 11-6 PIC18 定时器的中断标志位

中断	标志位(寄存器)	使能位(寄存器)	优先级(寄存器)
Timer0	TMR0IF(INTCON)	TMR0IE(INTCON)	TMR0IP(INTCON2)
Timer1	TMR1IF(PIR1)	TMR1IE(PIE1)	TMR1IP(IPR1)
Timer2	TMR2IF(PIR1)	TMR2IE(PIE1)	TMR2IP(IPR1)
Timer3	TMR3IF(PIR3)	TMR3IE(PIE2)	TMR3IP(IPR2)
INT1	INT1IF(PIR1)	INT1IE(PIE1)	INT1IP(INTCON3)

（续）

中　断	标志位(寄存器)	使能位(寄存器)	优先级(寄存器)
INT2	INT2IF(PIR1)	INT2IE(PIE1)	INT2IP(INTCON)
TXIF	TXIF(PIR1)	TXIE(PIE1)	TXIP(IPR1)
RCIF	RCIF(PIR1)	RCIE(PIE1)	RCIP(IPR1)
RB_INT	RBIF(INTCON)	RBIE(INTCON)	RBIP(INTCON2)

455

注意：INTO 只有高优先级。

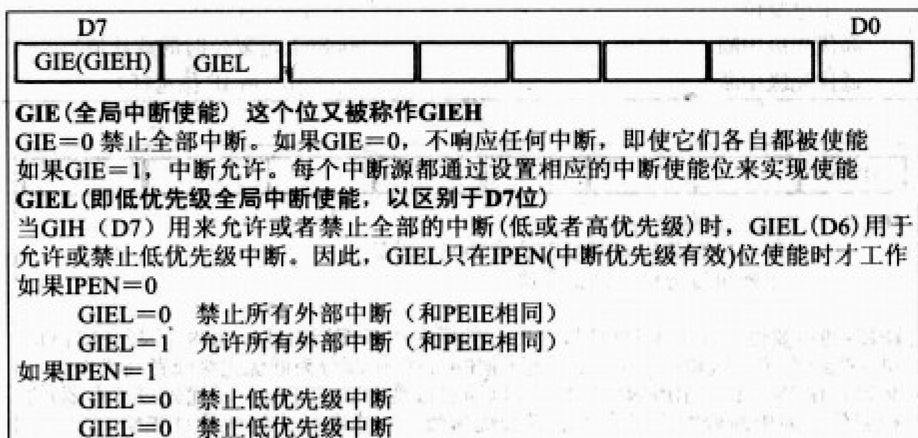
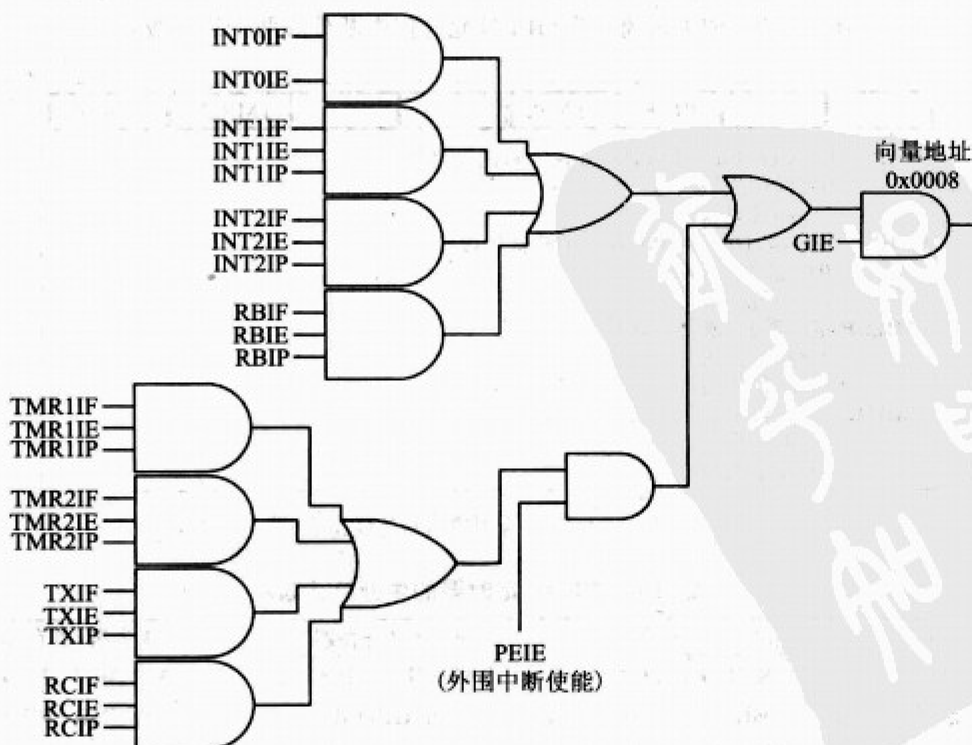


图 11-21 INTCON(中断控制)寄存器



456

图 11-22 带高优先级(IP)标志位的中断

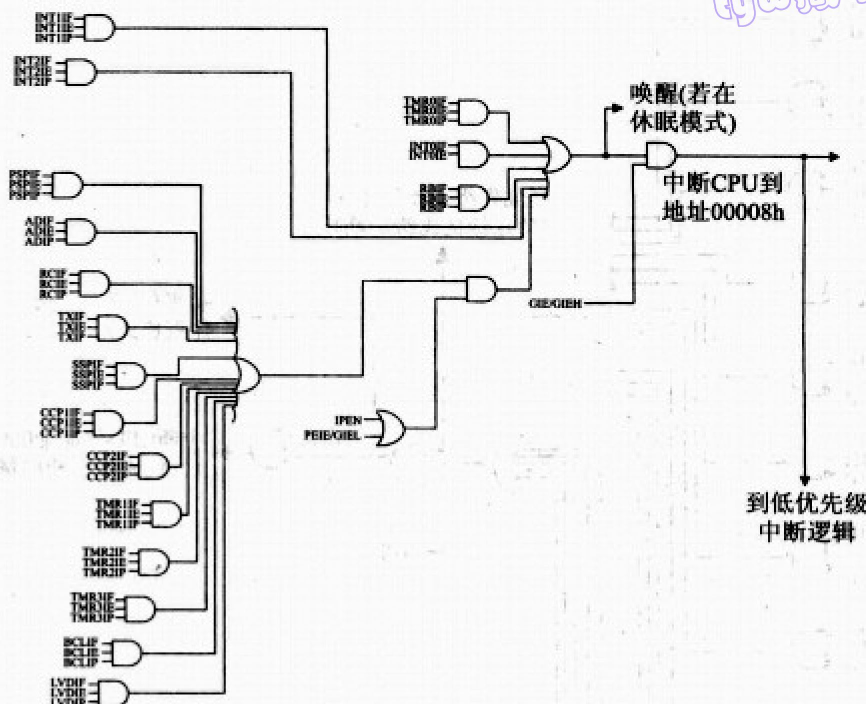


图 11-23 高优先级中断(摘自 PIC18 手册)

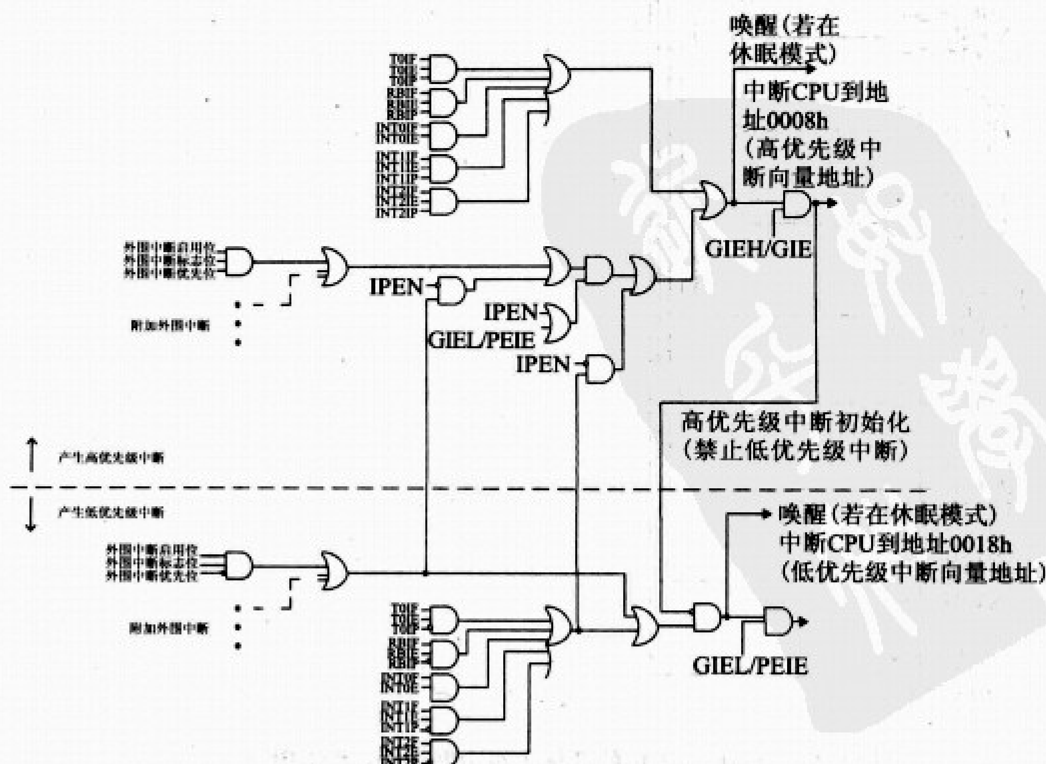


图 11-24 低优先级和高优先级中断选择(摘自 PIC18 手册)

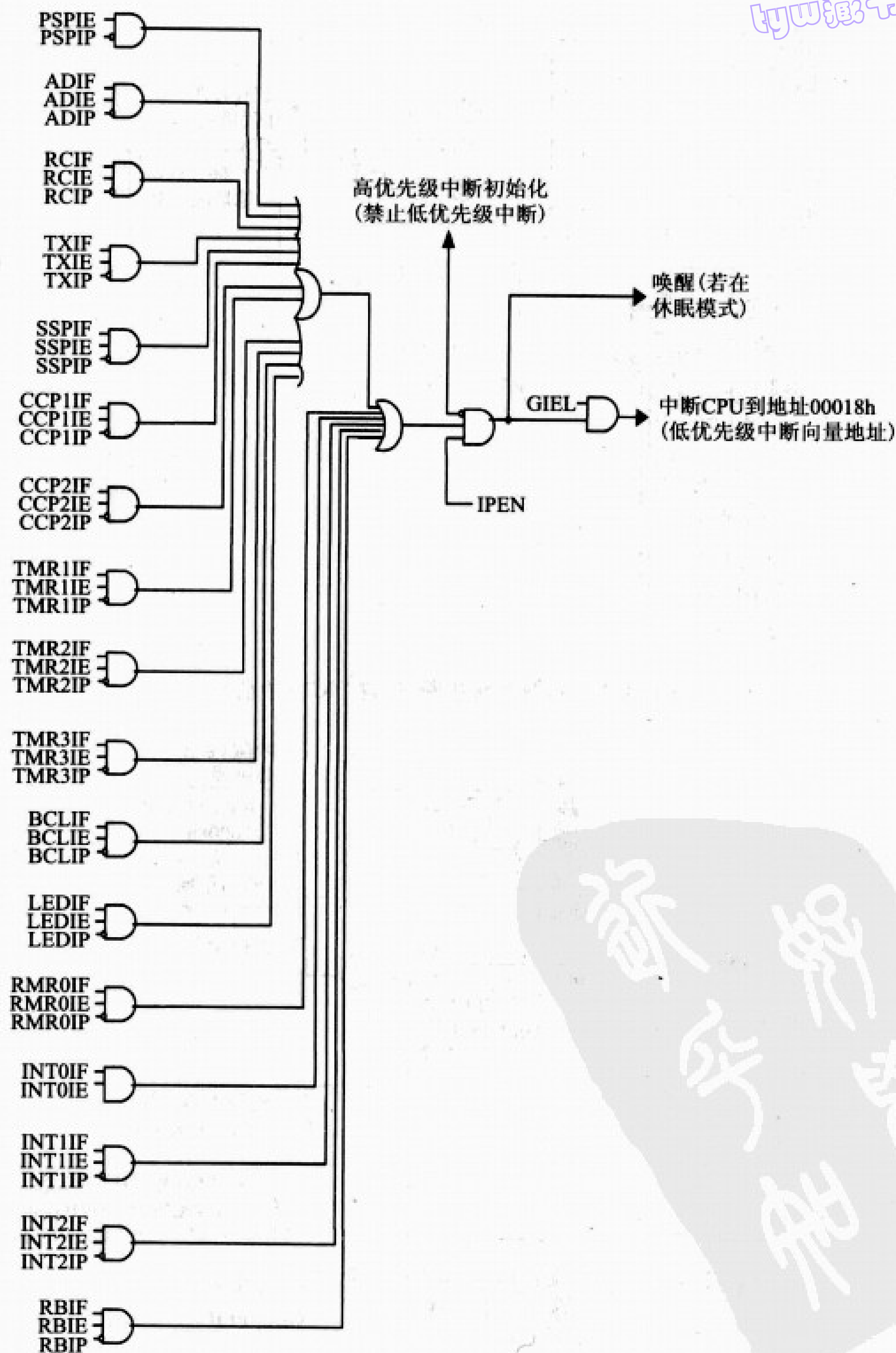


图 11-25　带 IP 标志位的低优先级中断选择(摘自 PIC18 手册)

程序 11-10 使用定时器 0 和定时器 1 中断分别在引脚 RB1 和 RB7 产生方波，同时将数据从 PORTC 传送到 PORTD。这是程序 11-2 的重现，只不过对定时器 1 分配了低优先级而已。

程序 11-10

```
;Program 11-10
      ORG   0000H
      GOTO  MAIN        ;bypass interrupt vector table
;--high-priority interrupts go to address 00008
      ORG   0008H       ;high-priority interrupt vector table
      BTFSC INTCON,TMR0IF   ;Is it Timer0 interrupt?
      BRA T0_ISR            ;Yes. Then branch to T0_ISR
      RETFIE 0x01           ;No. Then fast return to main
;--low-priority interrupts go to address 00018
      ORG   0018H       ;low-priority interrupt vector table
      BTFSC PIR1,TMR1IF     ;Is it Timer1 interrupt?
      BRA T1_ISR            ;Yes. Then branch to T1_ISR
      RETFIE                ;No. Then return to main

;-main program for initialization and keeping CPU busy
      ORG   0100H ;somewhere after vector table space
MAIN  BCF   TRISB,1     ;PB1 as an output
      BCF   TRISB,7     ;PB7 as an output
      CLRF  TRISD       ;make PORTD output
      SETF  TRISC       ;make PORTC input
      MOVLW 0x08        ;Timer0, 16-bit, no prescale,
                        ;internal clk
      MOVWF T0CON       ;load T0CON reg
      MOVLW  0xFF       ;TMR0H = FFH, the high byte
      MOVWF  TMR0H      ;load Timer0 high byte
      MOVLW 0x00        ;TMR0L = 00H, the low byte
      MOVWF TMR0L       ;load Timer0 low byte
      BCF INTCON,TMR0IF ;clear Timer0 interrupt flag bit
      BSF INTCON,TMR0IE ;enable Timer0 interrupt
      MOVLW  0x0        ;Timer1, 16-bit, no prescale,
                        ;internal clk
      MOVWF T1CON       ;load T1CON reg
      MOVLW  0xFF       ;TMR1H = FFH, the high byte
      MOVWF  TMR1H      ;load Timer0 high byte
      MOVLW 0x00        ;TMR1L = 00H, the low byte
      MOVWF TMR1L       ;load Timer1 low byte
      BCF PIR1,TMR1IF   ;clear Timer1 interrupt flag bit
      BSF PIE1,TMR1IE   ;enable Timer1 interrupt
      BCF IPR1,TMR1IP   ;make Timer1 low-priority interrupt
      BSF   RCON,IPEN   ;enable priority levels
      BSF INTCON,GIEL
      BSF INTCON,GIEH   ;enable interrupts globally
      BSF T1CON,TMR1ON;start Timer1
      BSF T0CON,TMR0ON;start Timer0
;--keeping CPU busy waiting for interrupt
OVER  MOVFF PORTC,PORTD ;send data from PORTC to PORTD
      BRA OVER          ;stay in this loop forever
;--------------------------ISR for Timer0
T0_ISR
      ORG 200H
```

```
        MOVLW  0xFF          ;TMR0H = FFH, the high byte
        MOVWF  TMR0H         ;load Timer0 high byte
        MOVLW 0x00           ;TMR0L = 00H, the low byte
        MOVWF  TMR0L         ;load Timer0 low byte
        BTG   PORTB,1        ;toggle RB1
        BCF  INTCON,TMR0IF ;clear timer interrupt flag bit
        RETFIE 0x01
;--------------------------ISR for Timer1
T1_ISR
        ORG 300H
        MOVLW  0xFF          ;TMR1H = FFH, the high byte
        MOVWF  TMR1H         ;load Timer0 high byte
        MOVLW 0x00           ;TMR1L = 00H, the low byte
        MOVWF  TMR1L         ;load Timer1 low byte
        BTG PORTB,7
        BCF PIR1,TMR1IF ;clear Timer1 interrupt flag bit
        RETFIE
        END
```

程序 11-11 有 4 个中断。它使用定时器 0 和定时器 1 中断在引脚 RC0 和 RC1 分别产生方波。它还使用发送和接收中断来发送和接收串行数据。数据从 PORTD 送出，在 PORTB 接收。定时器 0 和接收的 ISR 是高优先级中断，而定时器 1 和发送的 ISR 分配的是低优先级中断。

程序　11-11

```
;Program 11-11
      ORG    0000H
      GOTO  MAIN          ;bypass interrupt_vector table
;--high-priority interrupts go to address 00008
      ORG    0008H        ;need to redirect because not
                          ;enough space
      GOTO CHK_HI_PRIO
;--no need to redirect because we have plenty of space
      ORG    00018        ;low-priority interrupt vector table
      BTFSC PIR1,TMR1IF       ;is it Timer1 interrupt?
      BRA    T1_ISR       ;Yes. Then branch to T1_ISR
      BTFSC PIR1,TXIF     ;Did we get here due to TxD?
      BRA    TX_ISR       ;Yes. Then branch to TX_ISR
      RETFIE              ;No. Then return to main
;----------------
CHK_HI_PRIO      ORG    0x50
      BTFSC INTCON,TMR0IF   ;Is it Timer0 interrupt?
      BRA    T0_ISR       ;Yes. Then branch to T0_ISR
      BTFSC PIR1,RCIF     ;Did we get here due to RCV int?
      BRA RC_ISR          ;Yes. Then branch to RC_ISR
      RETFIE 0x01         ;No. Then return to main
 ;-main program for initialization and keeping CPU busy
      ORG    0100H        ;somewhere after vector table space
 MAIN BCF    TRISC,RC0
      BCF    TRISC,RC1
      CLRF TRISB          ;make PORTB output
      SETF TRISD          ;make PORTD input
      MOVLW 0x08          ;Timer0, 16-bit, no prescale,
```

```
                        ;internal clk
    MOVWF  T0CON        ;load T0CON reg
    MOVLW   0xFF        ;TMR0H = FFH, the high byte
    MOVWF   TMR0H       ;load Timer0 high byte
    MOVLW 0x00          ;TMR0L = 00H, the low byte
    MOVWF  TMR0L        ;load Timer0 low byte
    BCF INTCON,TMR0IF ;clear Timer0 interrupt flag bit
    MOVLW  0x0          ;Timer1, 16-bit, no prescale,
                        ;internal clk
    MOVWF  T1CON        ;load T1CON reg
    MOVLW   0xFF        ;TMR1H = FFH, the high byte
    MOVWF   TMR1H       ;load Timer0 high byte
    MOVLW 0x00          ;TMR1L = 00H, the low byte
    MOVWF  TMR1L        ;load Timer1 low byte
    BCF PIR1,TMR1IF   ;clear Timer1 interrupt flag bit
    MOVLW  0x20         ;enable transmit and choose low baud
    MOVWF  TXSTA        ;write to reg
    MOVLW  D'15'        ;9600 bps (Fosc / (64 * Speed) - 1)
    MOVWF  SPBRG        ;write to reg
    MOVLW  0x90         ;enable receive and serial port
    MOVWF RCSTA         ;write to reg
    BCF TRISC, TX       ;make TX pin of PORTC an output pin
    BSF TRISC, RX       ;make RCV pin of PORTC an input pin
    BSF   RCON,IPEN
    BSF   PIE1,RCIE
    BSF PIE1,TXIE       ;enable TX interrupt
    BSF INTCON,TMR0IE       ;enable Timer0 interrupt
    BSF PIE1,TMR1IE   ;enable Timer1 interrupt
    BCF IPR1,TMR1IP   ;make Timer1 a low-priority interrupt
    BCF IPR1,TXIP  ;make Transmit a low-priority interrupt
    BSF T0CON,TMR0ON        ;start Timer0
    BSF T1CON,TMR1ON        ;start Timer1
    BSF INTCON,GIEL         ;enable low-priority interrupts
    BSF INTCON,GIEH         ;enable high-priority interrupts
;--keeping CPU busy waiting for interrupt
OVER BRA     OVER           ;stay in this loop forever

;--------------------------ISR for Timer0
T0_ISR     ORG 200H
    MOVLW   0xFF        ;TMR0H = FFH, the high byte
    MOVWF   TMR0H       ;load Timer0 high byte
    MOVLW 0x00          ;TMR0L = 00H, the low byte
    MOVWF  TMR0L        ;load Timer0 low byte
    BTG    PORTC,0      ;toggle RB1
    BCF INTCON,TMR0IF ;clear timer interrupt flag bit
    RETFIE 0x01
;--------------------------ISR for Timer1
T1_ISR     ORG 300H
    MOVLW   0xFF        ;TMR1H = FFH, the high byte
    MOVWF   TMR1H       ;load Timer0 high byte
    MOVLW 0x00          ;TMR1L = 00H, the low byte
    MOVWF  TMR1L        ;load Timer1 low byte
    BTG PORTC,1
```

```
      BCF PIR1,TMR1IF ;clear Timer1 interrupt flag bit
      RETFIE
;----------------Transmit ISR
TX_ISR
      BCF PIR1,TXIF ;clear TX interrupt flag bit
      MOVFF PORTD,TXREG
      RETFIE
;-------------
RC_ISR
      MOVFF RCREG,PORTB   ;copy received data to PORTD
      BCF   PIR1, RCIF       ;clear RCIF
      RETFIE 1
      END
```

例 11-3 在程序 11-11(或者是程序 11-11C)中,讨论下面的情况:(a)如果在 PIC18 执行定时器 1 中断的时候,RCIF 中断被激发;(b)在定时器 0 中断被服务的时候,定时器 1 中断被激发;(c)RCIF 和 TMR0IF 以及 TMR1IF 同时被激发。

解:

在程序 11-11 中,注意接收标志(RCIF)和定时器 0(TMR0IF)中断是高优先级中断,而发送标志(TXIF)和定时器 1(TMR1IF)中断是低优先级的。因此,可以做如下的推断。

(1) 如果 RCIF 在定时器 1 的 ISR 执行期间被激活,那么接收中断进入,因为高优先级将首先执行它的 ISR。当 ISR 执行完后,PIC18 再返回去完成定时器 1 的 ISR。

(2) 如果在定时器 0 的 ISR 执行期间 TMR1IF 被激活,那么它将被忽略,因为它是低优先级的。在定时器 0 的 ISR 完成后,PIC18 才会执行定时器 1 的 ISR。

(3) 如果 3 个中断(RCIF、TMR0IF 和 TMR1IF)同时被激发,接收 ISR 和定时器 0 的 ISR 将首先被处理,因为它们拥有高优先级。在接收和定时器 0 中断之间,将因为中断向量表中的高优先级中断设置而先处理定时器 0 的 ISR。也就是说,在同时触发这 3 个中断时,它们的执行顺序依次是:定时器 0 的 ISR、接收 ISR 和定时器 1 的 ISR。

462

11.6.2 低优先级中断的 C 编程

正如在前 4 节中看到的,C18 编译器使用保留的关键字 **interrupt** 来指定中断为高优先级。为了给中断分配低优先级,可以使用关键字 **interruptlow**。如表 11-7 所示。请参阅程序 11-11C,它是程序 11-11 的 C 语言版本。

表 11-7 使用 C18 的 PIC18 中断向量表

中 断	ROM 地址	C18 关键字
高优先级中断	0x0008(默认)	**interrupt**
低优先级中断	0x0018(使用 IP 位来选择)	**interruptlow**

程序 11-11C 有 4 个中断。它使用定时器 0 和定时器 1 中断在引脚 RC0 和 RC1 分别产生方波。此外,还使用发送和接收中断来发送和接收串行数据。数据通过 PORTD 送出,在 PORTB 接收。定时器 0 和接收的 ISR 是高优先级中断,而定时器 1 和发送的 ISR 分配的是低优先级中断。

程序 11-11C

```
//Program 11-11C (This is the C version of Program 11-11)
#include <p18F458.h>
#define myPC0bit PORTCbits.RC0
#define myPC1bit PORTCbits.RC1

void chk_isr(void);
void chk_low_isr(void);
void T0_ISR(void);
void T1_ISR(void);
void TX_ISR(void);
void RC_ISR(void);

#pragma code My_HiPrio_Int =0x0008 //high-priority int
void My_HiPrio_Int (void)
{
_asm
GOTO chk_isr
_endasm
}

#pragma code My_Lo_Prio_Int =0x00018 //low-priority int
void My_Lo_Prio_Int (void)
{
    _asm
        GOTO chk_low_isr
    _endasm
}

#pragma interruptlow chk_low_isr //used for low-priority
void chk_low_isr (void)
{
    if(PIR1bits.TMR1IF==1)//Timer1 causes interrupt?
      T1_ISR();           //Yes. Execute Timer1 ISR
    if (PIR1bits.TXIF==1) //Transmit causes interrupt?
      TX_ISR( );          //Yes. Execute Transmit ISR
}

  #pragma interrupt chk_isr//used for high-priority interrupt
   void chk_isr (void)
{
     if (PIR1bits.TMR1IF==1)//Timer0 causes interrupt?
          T0_ISR( );        //Yes. Execute Timer0 ISR
     if (PIR1bits.RCIF==1) //Receiver causes interrupt?
          RC_ISR( );        //Yes. Execute Receiver ISR
}

void main(void)
  {
     TRISCbits.TRISC0=0;   //RC0 = OUTPUT
     TRISCbits.TRISC1=0;   //RC1 = OUTPUT
     TRISD = 255;          //PORTD = INPUT
     TRISB = 0;            //PORTB = OUTPUT
```

```
    T0CON=0x08;              //Timer0, 16-bit mode,
                             //no prescaler
    TMR0H=0xFF;              //load TH0
    TMR0L=0x00;              //load TL0
    INTCONbits.TMR0IF=0;     //clear TF1
    T1CON=0x0; //Timer 1, 16-bit mode, no prescaler
    TMR1H=0xFF;              //load TH1
    TMR1L=0x00;              //load TL1
    PIR1bits.TMR1IF=0;       //clear TF1
    TXSTA=0x20;              //choose low baud rate,8-bit
    SPBRG=15;                //9600 baud rate/ XTAL = 10 MHz
    RCSTAbits.CREN=1;
    RCSTAbits.SPEN=1;
    TRISCbits.TRISC6=0;      //TX pin = OUTPUT
    TRISCbits.TRISC7=1;      //RCV pin = INPUT
    RCONbits.IPEN=1;
    PIE1bits.RCIE=1;
    PIE1bits.TXIE=1;         //enable TX interrupt
    INTCONbits.TMR0IE=1;
    PIE1bits.TMR1IE=1;
    IPR1bits.TMR1IP=0;       //make Timer1 a low-priority
    IPR1bits.TXIP=0;         //make TX a low-priority
    T0CONbits.TMR0ON=1;      //turn on T0
    T1CONbits.TMR1ON=1;      //turn on T1
    INTCONbits.GIEL=1;       //enable low-priority interrupts
    INTCONbits.GIEH=1;//enable high-priority interrupts
    while(1);          //keep looping until interrupt comes
  }
//-------------ISR for Timer0
void T0_ISR(void)
  {
    TMR0H=0xFF;              //load TH0
    TMR0L=0x00;              //load TL0
    myPC0bit=~myPC0bit;      //toggle RB1
    INTCONbits.TMR0IF=0;     //clear TF0
  }
//-------------ISR for Timer1
void T1_ISR(void)
  {
    TMR1H=0xFF;              //load TH0
    TMR1L=0x00;              //load TL0
    PIR1bits.TMR1IF=0;       //clear TF1
    myPC1bit=~myPC1bit;      //toggle RB1
  }
//-------------ISR for Transmit
void TX_ISR(void)
  {
   TXREG=PORTD;        //clear Tx Interrupt flag
  }
//-------------ISR for Receive
void RC_ISR(void)
  {
    PORTB=RCREG;
```

464

```
    RCSTAbits.CREN=0;//clear CREN to clear any error
    RCSTAbits.CREN=1;//set CREN for continuous reception
}
```

11.6.3 中断嵌套

当PIC18正在执行一个中断的ISR时,另一个中断被触发,会发生什么呢?在这种情况下,高优先级中断可以打断低优先级中断。这就是中断嵌套。PIC18的低优先级中断可以被高优先级中断打断,但不会被另一个低优先级中断打断。虽然所有的中断都是锁存的并且内部保存的,但是低优先级中断都不能立即得到CPU的应答,直到PIC18完成所有的高优先级中断。GIE位(也叫作GIEH)和GIEL位在中断嵌套的处理中有重要的作用。在考虑嵌套中断的时候,必须注意以下几点。

(1) 当一个高优先级中断向量指向地址0008H时,GIE位是被禁用的(GIEH=0),因此就阻止了其他中断(低优先级或者高优先级)的进入。ISR的最后一条指令RETFIE会自动地使能GIE位(GIE=1),从而允许其他中断的进入。如果希望在执行当前ISR的过程中就让另一个高优先级中断进入,那么必须在当前ISR的起始处将GIE置1。

(2) 当一个低优先级中断向量指向地址0018H时,GIEL位是被禁用的(GIEL=0),因此就阻止了其他低优先级中断的进入。ISR的最后一条指令RETFIE会自动地使能GIEL位(GIEL=1),从而允许其他中断的进入。注意,在执行当前低优先级中断时,不能阻止高优先级中断的进入,因为GIEH仍然是置为1的(GIEH=1)。

465

(3) 当两个或者更多的中断具有相同的优先级时,程序会按照在中断向量表中的查询顺序来执行。在本章中已经看到了很多这样的例子。请参阅例11-3。

11.6.4 在任务转换时变量的快速保存

在很多应用[例如多任务实时操作系统(RTOS)]中,CPU每次读入一个任务(工作或者进程)并执行,然后再转到下一个任务。在执行每个任务的时候(即执行中断服务程序时),及时地访问CPU的所有资源是非常重要的。在早期的CPU中,有限数量的寄存器迫使程序员来执行新的任务之前保存栈中CPU的内容。这种在执行新任务之前保存CPU内容的方式,叫作变量保存(变量切换)。使用栈来保存CPU内容既麻烦,又浪费时间,而且速度又慢。因此,一些CPU[如x86微处理器的指令PUSHA(全部压栈)和POPA(全部出栈)]使用一条指令来实现把所有主要寄存器的内容压栈或出栈。因为PIC18的通用寄存器数量很多,所以不需要使用栈来保存CPU的通用寄存器内容。然而,每个任务都会用到WREG寄存器、BSR寄存器和STATUS寄存器这几个主要的寄存器。因此,当高优先级中断被激活时,PIC18内部自动把这些寄存器的内容保存在影子寄存器中。按照这种方式,主任务的3个关键寄存器就由内部保存。要恢复这3个寄存器的原有内容,必须在高优先级ISR的出口处使用指令RETFIE 0x01来代替指令REIFIE。RETFIE 0x01在PIC18的文献中被叫作快速变量保存。在PIC18中使用快速变量保存,要注意下面的两点。

(1) 它对低优先级中断是无效的,只对高优先级中断起作用。也就是说,当一个低优先级中断被激活时,就不存在快速变量保存。如果低优先级的ISR用到这3个寄存器的内容,必

须在低优先级的ISR开始处时保存这些寄存器的内容。

(2) 保存这3个关键寄存器的影子寄存器只有一个寄存器的容量，因此只能用来保存一个寄存器的内容。基于这个原因，快速变量保存只有在主子例程的高优先级ISR被激活时才工作。当2个或者3个高优先级中断同时被激活时，只有第一个ISR能使用快速变量保存，因为影子寄存器只有一个容量。所以，第二个和第三个ISR必须在它们的ISR开始处保存关键寄存器的内容。在知道了ISR的执行顺序后，这个工作并不难。也可以参阅本章中的许多例子。

466

11.6.5 中断延迟

从一个中断被激活，到CPU开始执行向量表地址0008H(或者是0x0018H)中的代码，这段时间叫作*中断延迟*。该延迟可以是2个到4个指令不等的时钟周期，这取决于中断源是内部的(如定时器)还是外部硬件(如硬件INTx和PORTB-变化)中断。中断延迟的持续时间也会受到在中断到来时CPU正在执行的指令类型的影响。如果正在执行的指令是持续2个指令周期(如MOVFF reg,reg)，那么相比于持续1个指令周期的指令(如ADDWL)，这个时间会略微长一点。请参阅PIC18的时序数据表。

11.6.6 软件触发中断

有时候，需要使用仿真的方法来测试ISR。这可以使用一条简单的指令把中断设为高电平，使PIC18跳转到中断向量表。例如，如果定时器1的TMR1IE位变为高电平，诸如`BSF INTCON, TMR1IF`这样的指令将中断PIC18当前的工作，并转到中断向量表。换句话说，这里不需要等到定时器1复零才中断。因此，可以使用让中断标志位变为高电平的指令来引起中断。

11.6.7 复习题

1. 判断对错：在上电复位时，所有的中断都有同等的优先级。
2. PIC18的哪个寄存器的哪一位是用来使能中断优先级的？它是一个位可寻址的寄存器吗？
3. TXIP位在哪个寄存器中？请说明如何为它分配低优先级。
4. 假设INT0和INT1都是低优先级。如果INT0和INT1同时被触发，将会发生什么情况呢？同时假设程序在中断向量表中将先查询INT0。
5. 当PIC18在执行一个低优先级中断(例如，正在执行一个低优先级ISR)时，一个高优先级中断被激发，这将会发生什么情况呢？

小结

中断是一个外部或者内部事件，用来打断微控制器当前的工作，并通知它某个设备需要处理。每个中断都有一个对应的程序，叫作ISR，或者是中断服务程序。PIC18有多个中断源，这取决于产品的型号。PIC18最常用的中断是定时器中断、外部硬件中断和串行通信中断。当一个中断被触发时，IF(中断标志位)会变为高电平。

PIC18可以通过对GIE(全局中断允许)和IE(中断允许)位编程，来使能(允许)或者禁止

(屏蔽)一个中断。PIC18 有两个优先级:低优先级和高优先级。在上电复位时,所有的中断都是高优先级的,并指向中断向量表中的地址 0008H。这个默认的设置可以通过 IP(中断优先级)位来改变。通过对 IP 位编程,可以把一个中断设为低优先级,并放入中断向量表的地址 0x00018。本章还介绍了如何使用汇编语言和 C 语言对 PIC18 进行中断编程。 467

习题

1. 在中断和查询方式中,哪一种方式可以避免绑定微控制器?
2. 请列出 PIC18 的一些中断源。
3. 在 PIC18 中,哪个存储空间是分配给中断向量表的?
4. 判断对错:PIC18 的程序员不能改变分配给中断向量表的存储地址。
5. 在中断向量表里,哪个地址是分配给低优先级中断的?
6. 在中断向量表里,哪个地址是分配给高优先级中断的?
7. 在中断向量表中,有分配给定时器 0 中断的存储地址吗?
8. 在中断向量表中,有分配给定时器 1 中断的存储地址吗?
9. GIE 位属于哪个寄存器?
10. 为什么要把 GOTO 指令放在地址 0?
11. 在上电复位时,GIE 位的状态是什么?其作用是什么呢?
12. 写出使能 INT0 中断的指令。
13. 写出使能定时器 0 中断的指令。
14. TMR0IE 位属于________寄存器。
15. 在中断向量表中,有多少字节的空间是分配给高优先级中断的?
16. 在中断向量表中,有多少字节的空间是分配给低优先级中断的?
17. 为了把中断服务程序放入中断向量表中高优先级的地址,则它的大小不能大于________字节。
18. 判断对错:INTCON 寄存器不是位可寻址的寄存器。
19. 请使用一条单指令,说明如何禁止所有中断。
20. 请使用一条单指令,说明如何禁止 INT0 中断。
21. 判断对错:在上电复位时,PIC18 允许所有的中断。
22. 在 PIC18 中,有多少字节的 ROM 空间是分配给复位的?
23. 判断对错:对于定时器 0 和定时器 1,在中断向量表中各自有对应的唯一地址。
24. 在中断向量表中,哪个地址是分配给定时器 1 的?
25. 请说明如何使能定时器 0 中断。
26. INTCON 寄存器的哪一位属于定时器 0 中断?请说明如何启用它。 468
27. 假设定时器 0 被编程为 8 位模式,TMR0H=F0H,使能 TMR0IE 位。请说明定时器中断是如何工作的。
28. 判断对错:定时器 1 的 ISR 最后两条指令是:

```
BCF  PIR1,TMR1IF
RETFIE
```

29. 假设定时器 1 编程为 16 位模式,TMR1H=FFH,使能 TMR1IE 位。请说明中断是如何触发的。
30. 如果定时器 1 编程为 8 位模式,请说明中断是何时被触发的。
31. 编制程序,使用定时器 0 在 RB7 引脚产生频率为 1Hz 的方波,同时将数据从 PORTC 传送到

PORTD。假设 XTAL=10 MHz。

32. 编制程序，使用定时器 1 在 RB7 引脚产生频率为 3 kHz 的方波，同时将数据从 PORTC 传送到 PORTD。假设 XTAL=10 MHz。
33. 判断对错：每个外部硬件中断 INT0、INT1 和 INT2 都配有一个地址。
34. 在中断向量表中，哪个地址是分配给 INT0、INT1 和 INT2 的？在 PORTB 上的引脚数是多少呢？
35. INT0IE 位属于哪个寄存器？请说明如何使能它。
36. INT1IE 位属于哪个寄存器？请说明如何使能它。
37. 请说明如何使能所有的 3 个外部硬件中断。
38. 假设用于外部硬件中断的 INT0IE 位 INT0 已被使能，而且是下降沿触发。请说明该中断被触发时，中断是如何工作的。
39. 判断对错：在上电复位时，外部硬件中断都是设为下降沿触发的。
40. 在 38 题中，如何确保单个中断不会被当作多个中断呢？
41. INT0IF 位属于________寄存器。
42. INT1IF 位属于________寄存器。
43. 判断对错：INT1 的 ISR 最后两条指令是：

```
BCF  INTCON3, INT1IF
RETFIE
```

44. 请解释在执行外部中断 0 时 INT0IF 和 INT0IE 的作用。
45. 请解释在执行外部中断 1 时 INT1IF 和 INT1IE 的作用。
46. 假设用于外部硬件中断的 INT1IE 位 INT1 已被使能，而且是上升沿触发。请说明它被触发时，中断是如何工作的。它又是如何确保一个中断没有被当成多个中断呢？
47. 判断对错：INT0～INT2 是 PEIE 组的一部分。
48. 判断对错：在上电复位时，INT0～INT2 的所有中断都是设为上升沿触发的。

469

49. 请指出上升沿触发和下降沿触发之间的区别。
50. 如何把硬件中断设为下降沿触发？
51. 判断对错：INT0～INT2 必须配置为硬件中断的输入引脚。
52. INTEDGx 位属于哪个寄存器？
53. 判断对错：有两个独立的中断分配给中断 TXIF 和 RCIF。
54. 在上电复位时，中断向量表中的哪个地址是分配给串行中断的？共有多少个字节？
55. TXIF 属于哪个寄存器？请说明如何使能它。
56. 假设串行中断的 TXIE 位已被使能。请说明如何触发该中断，以及中断触发时的工作过程。
57. 判断对错：在上电复位时，串行中断是被禁用的。
58. 判断对错：发送中断的 ISR 最后两条指令是：

```
BCF  PIR1, TXIF
RETFIE
```

59. 请说明清零 RCIF 的过程。
60. 假设在 TXIF 为高电平时 TXIE 位为 1，接下来将会发生什么情况？
61. 假设在 RCIF 为高电平时 RCIE 位为 1，接下来将会发生什么情况？
62. 编制程序，使用中断把接收到的串行数据送入 PORTD，同时让连接到 PORTC.7 的 LED 翻转以反映 PORTB.4 的变化。

63. 请写出关于 PORTB 变化中断的下列信息。
   (a)与 PORTB 变化中断有关的标志位。
   (b)这些标志位所在的寄存器。
   (c)PORTB 变化中断和 INT0～INT2 中断之间的区别。
   (d)属于 PORTB 变化中断的引脚。
64. 判断对错:在上电复位时,所有中断都是高优先级的。
65. 哪个寄存器用来使能 PIC18 的中断优先级? 请解释它的作用。
66. INT0IP 位属于哪个寄存器? 请说明如何把它分配成低优先级。
67. TMR1IP 位属于哪个寄存器? 请说明如何把它分配成低优先级。
68. INT1IP 位属于哪个寄存器? 请说明如何把它分配成低优先级。
69. 假设 INT1IP 和 INT2IP 都是 0。如果 INT1IF 和 INT2IF 同时被激活,将会怎么样?
70. 假设 TMR0IP 和 TMR1IP 都是 0。如果 TMR0IF 和 TMR1IF 同时被激活,将会怎么样?
71. 假定 TMR0IP 和 TMR1IP 都设成高优先级,如果它们同时被激活,将会怎么样?
72. 假定 INT1IP 和 INT2IP 都设成高优先级,如果它们同时被激活,将会怎么样?
73. 当 PIC18 正在处理一个高优先级中断时,一个低优先级中断被激活,这将会怎么样?
74. 当 PIC18 正在处理一个低优先级中断时,一个高优先级中断被激活,这将会怎么样?
75. 解释 GIEH 位在允许和禁止中断时的作用。
76. 判断对错:对于 PIC18,不允许中断嵌套。
77. 解释 GIEL 位在允许和禁止中断时的作用。
78. 解释 RETFIE 在允许 GIEL 位时的作用。
79. 解释"RETFIE"和"RETFIE 1"指令之间的区别。
80. 解释 PIC 快速变量保存的概念。

470

复习题答案

11.1 节

1. 中断　　2. INT0 和 TMR0
3. 地址 0x0008～地址 0x00017。否。它是在处理器设计的时候就设置好的。
4. GIE=0 表明所有的中断都被屏蔽了,PIC18 不会响应任何的中断。
5. 假设 GIE=1,则需使用指令 BSF INTCON, TMR0IE。
6. 地址 0008 分配给高优先级中断,而地址 0x0018 分配给低优先级中断。

11.2 节

1. 错误。所有定时器(如定时器 0、定时器 1 等)都配有一个地址。
2. 0008H
3. PIE1,指令 BSF INTCON, TMR0IE 用来使能定时器 1 中断。
4. 在定时器 1 启动后,定时器从 F5H 计数到 FFH,而 PIC18 执行其他的任务。当从 FFH 跳变到 00 时,TMR1IF 位变为高电平,中断 PIC18 当前的任务,让它跳转到地址 0008 来执行这个中断的 ISR。
5. 正确。

11.3 节

1. 正确。

2. RB0(PORTB.0)、RB1(PORTB.1)和 RB2(PORTB.2)位。
3. BSF INTCON2, INT1IE
4. 当在引脚 RB0 施加由低到高的脉冲时,PIC18 中断当前的任务,跳转到 ROM 地址 0008H 处执行 ISR。
5. 错误。
6. 当 CPU 跳转到 ROM 地址 0008 处执行 ISR 时,GIE 变为 0,以阻止同一中断源的另一个中断。ISR 的最后两条指令是 BCF INCON, INT0IF 和 RETFIE。第一条指令会清除过去的中断请求,而第二条指令则置 GIE=1,允许该中断源的新中断。只有当新的由低到高脉冲信号施加到该引脚时,才会发生中断。

471

7. 正确。

11.4 节

1. 正确。对于所有的中断源(包括发送和接收),都只有一个中断。
2. 地址 0x0008 分配给高优先级中断,而地址 0x0018 分配给低优先级中断。
3. 指令 BSF PIE1, TXIE 允许发送中断,而指令 BSF PIE1,RCIE 允许接收中断。
4. 当接收完整帧的数据(包括结束位)时,RCIF(接收中断标志位)会变为高电平。然后将接收到的字节送入 RCREG 寄存器,PIC18 跳转到地址 0008H 处执行该中断对应的 ISR。在串行 COM 中断服务程序中,必须在新的数据写入之前将 RCREG 的内容保存起来。
5. 错误。　6. 正确。
7. BCF RIR1, TXIF
   RETFIE

11.5 节

1. 错误。　2. 所有的中断(包括 PORTB-变化中断)都指向默认地址 0008。
3. INTCON,可以使用指令 BSF INTCON , RBIF 来使能它。
4. BCF INTCON, RBIF
   RETFIE
5. 错误。

11.6 节

1. 正确。　2. RCON 寄存器的 IPEN 位。是的,它是位可寻址的。
3. IPR1 和指令 BCF IPR1, TXIP 可以完成这个工作。
4. 如果被同时激活,INT0 会先被响应,因为它先被查询到。在执行完 INT0 后会执行 INT1。
5. 一个中断里面嵌入新的中断,也就是说暂停执行低优先级中断,转而执行更高优先级的中断。在执行完该高优先级中断后,PIC18 会继续执行低优先级 ISR。

472

第 12 章 LCD 和键盘接口

学习目标：

- ☐典型 LCD 的引脚功能
- ☐用于 LCD 编程的指令代码
- ☐LCD 与 PIC18 的接口
- ☐LCD 的汇编语言和 C 语言编程
- ☐键盘的基础操作
- ☐按键和检测机制
- ☐PIC18 的 4×4 键盘的汇编和 C 编程

本章将讨论 PIC18 在实际生活中的一些应用。本章将会介绍 PIC18 和各种设备(如 LCD 和键盘)的连接。12.1 节主要讲述 LCD 和 PIC18 的接口，12.2 节主要讲述键盘和 PIC18 的接口。这两节都使用的是 C 语言和汇编语言编程。

12.1 LCD 接口

本节将首先描述 LCD 的操作模式，接着将介绍如何使用汇编语言和 C 语言对 LCD 和 PIC18 的接口进行编程。

12.1.1 LCD 操作

近几年来，LCD 已经被广泛地使用，并逐渐代替了 LED(七段 LED 或者其他的多段 LED)。这主要是因为以下几个原因。

(1) LCD 价格的下降。

(2) LCD 可以显示数字、字符和图像。相比之下，LED 则只能显示数字和一些字符。

(3) LCD 内置有刷新控制器，因此 CPU 可以从刷新 LCD 的工作中解放出来。相比之下，LED 必须通过 CPU(或使用其他的方法)来刷新以显示数据。

(4) LCD 更易于对字符和图像编程。

12.1.2 LCD 引脚描述

本节讨论的 LCD 有 14 个引脚。各个引脚的功能如表 12-1 所示。图 12-1 给出了各种 LCD 的引脚位置。

表 12-1 LCD引脚说明

引脚	符号	I/O	说明
1	V_{SS}	—	接地
2	V_{CC}	—	+5V电源
3	V_{EE}	—	控制对比度的电源
4	RS	I	RS=0，选择命令寄存器；RS=1，选择数据寄存器
5	R/W	I	R/W=0，用于写；R/W=1，用于读
6	E	I/O	使能端
7	DB0	I/O	8位数据总线
8	DB1	I/O	8位数据总线
9	DB2	I/O	8位数据总线
10	DB3	I/O	8位数据总线
11	DB4	I/O	8位数据总线
12	DB5	I/O	8位数据总线
13	DB6	I/O	8位数据总线
14	DB7	I/O	8位数据总线

1. V_{CC}、V_{SS}和V_{EE}

V_{CC}和V_{SS}分别提供5V电压和接地，V_{EE}用来控制LCD的对比度。

2. RS，寄存器选择端

在LCD中有2个非常重要的寄存器。RS引脚用来选择寄存器的类型。若RS=0，则选择指令命令代码寄存器，允许用户发送命令（如清除显示、光标复位等）。若RS=1，则选择数据寄存器，允许用户发送要在LCD上显示的数据。

3. R/W，读/写

R/W输入允许用户向LCD写信息或者从LCD读取信息。在读操作时R/W=1，在写操作时R/W=0。

4. E，使能端

使能端引脚被LCD用来锁存数据引脚上的信息。当数据传送到数据引脚时，为使LCD能锁存数据引脚上的数据，必须施加一个由高到低的脉冲到使能端。这个脉冲最小宽度为450 ns。在本书中，将该延迟时间称作SDELAY（短延时），以同其他的延迟相区别。

5. D0～D7

8位数据引脚（D0～D7），用来将信息传送到LCD或从LCD的内部寄存器读取信息。

为了显示字母和数字，当RS=1时，需要将字母A～Z、a～z及数字0～9的ASCII代码传送到这些引脚。

也可以发送一些指令命令代码来清除显示的内容，让光标复位或闪烁。表12-2列出了这些指令命令代码。为了向LCD发送表12-2中列出的命令，需要将RS设置为0。若是向LCD发送数据，则需要令RS=1，然后向使能端引脚发送一个由高到低的脉冲，以使能LCD的内部锁存器。向LCD发送字符（命令或者数据）的方法有两种：(1)

在发送下一个字符之前使用延迟；(2)使用 busy 标志来判断 LCD 是否准备好发送下一个字符。

表 12-2 LCD 命令代码

代码(十六进制)	发送到 LCD 指令寄存器的命令
1	清空显示屏
2	返回起始位置
4	光标减量(光标向左移)
6	光标增量(光标向右移)
5	显示画面右移
7	显示画面左移
8	显示关闭，光标关闭
A	显示关闭，光标打开
C	显示打开，光标关闭
E	显示打开，光标闪烁
F	显示打开，光标闪烁
10	光标移动到左端
14	光标移动到右端
18	将整个的显示画面左移
1C	将整个的显示画面右移
80	强制光标置于第一行的开头
C0	强制光标置于第二行的开头
38	二行和 5×7 矩阵

注意：该表源自表 12-4。

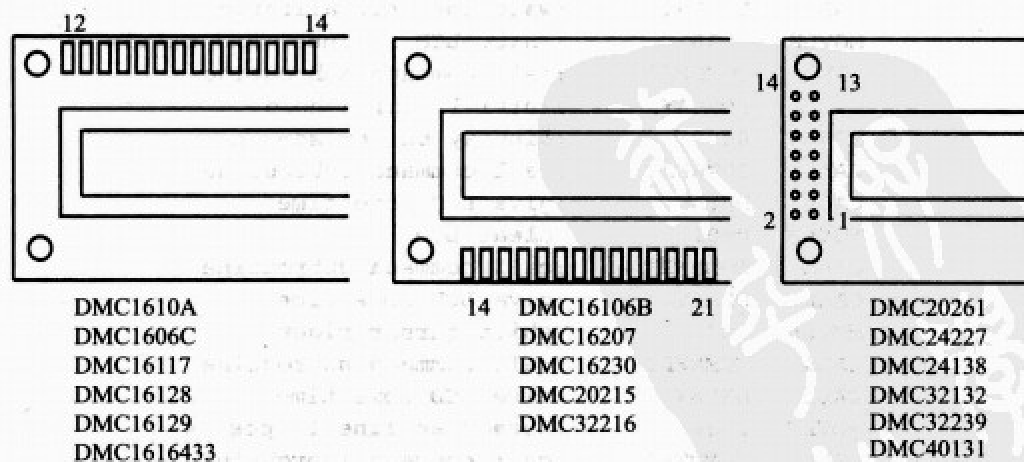


图 12-1 来自 Optrex 的各种 LCD 的引脚位置

12.1.3 为 LCD 发送带时间延迟的命令和数据

程序 12-1 说明了如何在不检测 busy 标志位的情况下向 LCD 发送字符(命令/数据)。注意，每向 LCD 发送一个字符就须等待 5ms～10ms(延迟)。这个延迟称为 DELAY。在进行 LCD 编程时，对于上电过程通常还需要一个长的延迟。这个延迟称为 LDELAY。SDELAY(短延迟)用于得到足够宽的使能端信号，以用于使能 LCD

输入。关于时延，请参阅第 3 章。

图 12-2 给出了 LCD 和微控制器的连接。

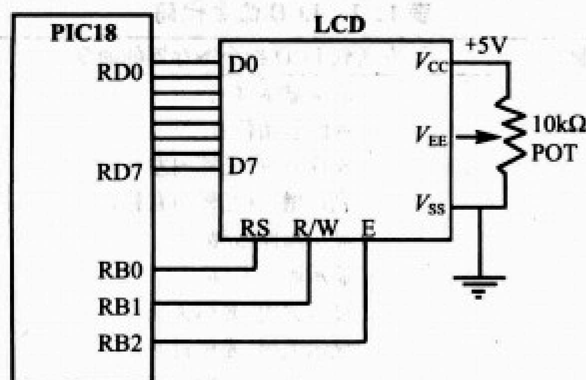


图 12-2 LCD 的连接

程序 12-1

```
;Program 12-1: Using delay before sending data/command
LCD_DATA    EQU PORTD           ;LCD data pins RD0-RD7
LCD_CTRL    EQU PORTB           ;LCD control pins
RS          EQU RB0             ;RS pin of LCD
RW          EQU RB1             ;R/W pin of LCD
EN          EQU RB2             ;E pin of LCD
            CLRF   TRISD        ;PORTD = Output
            CLRF   TRISB        ;PORTB = Output
            BCF    LCD_CTRL,EN  ;enable idle low
            CALL   LDELAY       ;wait for initialization
            MOVLW  0x38         ;init. LCD 2 lines, 5x7 matrix
            CALL   COMNWRT      ;call command subroutine
            CALL   LDELAY       ;initialization hold
            MOVLW  0x0E         ;display on, cursor on
            CALL   COMNWRT      ;call command subroutine
            CALL   DELAY        ;give LCD some time
            MOVLW  0x01         ;clear LCD
            CALL   COMNWRT      ;call command subroutine
            CALL   DELAY        ;give LCD some time
            MOVLW  0x06         ;shift cursor right
            CALL   COMNWRT      ;call command subroutine
            CALL   DELAY        ;give LCD some time
            MOVLW  0x84         ;cursor at line 1, pos. 4
            CALL   COMNWRT      ;call command subroutine
            CALL   DELAY        ;give LCD some time
            MOVLW  A'N'         ;display letter 'N'
            CALL   DATAWRT      ;call display subroutine
            CALL   DELAY        ;give LCD some time
            MOVLW  A'O'         ;display letter 'O'
```

```
            CALL     DATAWRT       ;call display subroutine
AGAIN       BTG      LCD_CTRL,0
            BRA      AGAIN         ;stay here
COMNWRT                            ;send command to LCD
            MOVWF    LCD_DATA      ;copy WREG to LCD DATA pin
            BCF      LCD_CTRL,RS   ;RS = 0 for command
            BCF      LCD_CTRL,RW   ;R/W = 0 for write
            BSF      LCD_CTRL,EN   ;E = 1 for high pulse
            CALL     SDELAY        ;make a wide En pulse
            BCF      LCD_CTRL,EN   ;E = 0 for H-to-L pulse
            RETURN
DATAWRT                            ;write data to LCD
            MOVWF    LCD_DATA      ;copy WREG to LCD DATA pin
            BSF      LCD_CTRL,RS   ;RS = 1 for data
            BCF      LCD_CTRL,RW   ;R/W = 0 for write
            BSF      LCD_CTRL,EN   ;E = 1 for high pulse
            CALL     SDELAY        ;make a wide En pulse
            BCF      LCD_CTRL,EN   ;E = 0 for H-to-L pulse
            RETURN
;look in previous chapters for delay routines
            END
```

12.1.4 使用 busy 标志位向 LCD 发送命令或数据

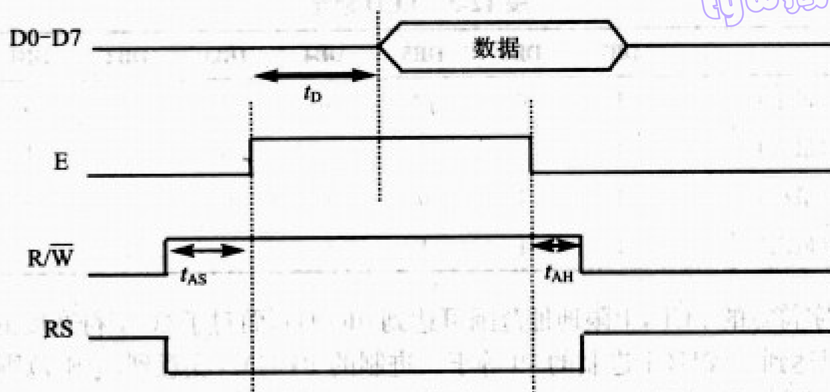
令 RS=0 读入 busy 标志位的状态，可以判断 LCD 是否准备好接收信息。Busy 标志位是 D7 位，在 R/W=1 和 RS=0 时，可以对其进行读操作，即 R/W=1，RS=0。D7=1(busy 标志位=1)表明 LCD 正进行内部操作，无法接收任何新的数据；D7=0 表明 LCD 可以接收新的数据。具体可以参阅程序 12-2。

程序 12-2

```
;Program 12-2: Check busy flag before sending
;data or command to LCD (See Fig. 12-2)
LCD_DATA    EQU PORTD              ;LCD data pins RD0-RD7
LCD_CTRL    EQU PORTB              ;LCD control pins
RS          EQU RB0                ;RS pin of LCD
RW          EQU RB1                ;R/W pin of LCD
EN          EQU RB2                ;E pin of LCD
            CLRF     TRISD         ;PORTD = Output
            CLRF     TRISB         ;PORTB = Output
            BCF      LCD_CTRL,EN   ;enable idle low
            CALL     LDELAY        ;long delay (250 ms) for power-up
            MOVLW    0x38          ;init. LCD 2 lines, 5x7 char
            CALL     COMMAND       ;issue command
            CALL     LDELAY        ;initialization hold
            MOVLW    0x0E          ;LCD on, cursor on
            CALL     COMMAND       ;issue command
            CALL     READY         ;Is LCD ready?
            MOVLW    0x01          ;clear LCD command
            CALL     COMMAND       ;issue command
            CALL     READY         ;Is LCD ready?
            MOVLW    0x06          ;shift cursor right
```

```
          CALL    COMMAND       ;issue command
          CALL    READY         ;Is LCD ready?
          MOVLW   0x86          ;cursor: line 1, pos. 6
          CALL    COMMAND       ;command subroutine
          CALL    READY         ;Is LCD ready?
          MOVLW   A'N'          ;display letter 'N'
          CALL    DATA_DISPLAY
          CALL    READY         ;Is LCD ready?
          MOVLW   A'O'          ;display letter 'O'
          CALL    DATA_DISPLAY
HERE      BRA     HERE          ;STAY HERE
;-----------------------------------------------
COMMAND   MOVWF   LCD_DATA      ;issue command code
          BCF     LCD_CTRL,RS   ;RS = 0 for command
          BCF     LCD_CTRL,RW   ;R/W = 0 for write
          BSF     LCD_CTRL,EN   ;E = 1 for high pulse
          CALL    SDELAY        ;make a wide En pulse
          BCF     LCD_CTRL,EN   ;E = 0 for H-to-L pulse
          RETURN
;-----------------------------------------------
DATA_DISPLAY MOVWF LCD_DATA     ;copy WREG to LCD DATA pin
          BSF     LCD_CTRL,RS   ;RS = 1 for data
          BCF     LCD_CTRL,RW   ;R/W = 0 for write
          BSF     LCD_CTRL,EN   ;E = 1 for high pulse
          CALL    SDELAY        ;make a wide En pulse
          BCF     LCD_CTRL,EN   ;E = 0 for H-to-L pulse
          RETURN
;-----------------------------------------------
READY     SETF    TRISD         ;make PORTD input port for LCD data
          BCF     LCD_CTRL,RS   ;RS = 0 access command reg
          BSF     LCD_CTRL,RW   ;R/W = 1 read command reg
;read command reg and check busy flag
BACK      BSF     LCD_CTRL,EN   ;E = 0 for L-to-H pulse
          CALL    SDELAY        ;make a wide En pulse
          BCF     LCD_CTRL,EN   ;E = 1 L-to-H pulse
          BTFSC   LCD_DATA,7    ;stay until busy flag = 0
          BRA     BACK
          CLRF    TRISD ;make PORTD output port for LCD data
          RETURN
;look in previous chapters for delay routines
          END
```

值得注意的是，在程序 12-2 中，busy 标志位是命令寄存器的 D7 位。为了读命令寄存器，令 R/W=1 及 RS=0，在使能引脚上施加一个由低到高的脉冲就可以得到命令寄存器。在读取命令寄存器后，如果 D7 位(busy 标志位)为 1，那么 LCD 处于忙状态，应该没有信息(命令/数据)可以发送给它。只有在 D7=0 时，LCD 才可以接受数据或命令。注意，使用这种方法不需要任何延时，因为在向 LCD 发送命令或数据之前都要先检测 busy 标志位的。对比图 12-3 和图 12-4 中读和写操作的时序。注意在 E 时序行中，写操作是下降沿触发的，而读操作是上升沿触发的。



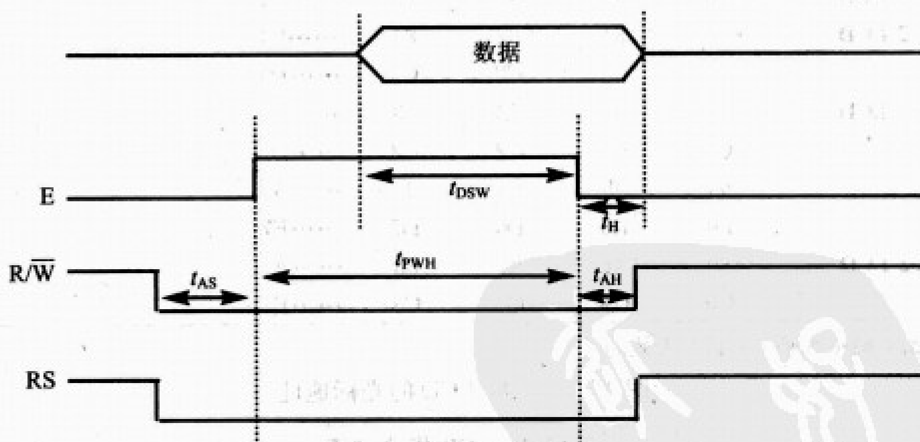
t_D=数据输出延迟时间

t_{AS}=在E信号之前，RS和R/W的启动时间=140 ns(最小)

t_{AH}=在E信号之后，RS和R/W的保持时间=10 ns(最小)

注意：读信号需要给E引脚一个由低到高的脉冲。

图 12-3　LCD 的读操作时序图(E 时序行电平从低到高)



t_{PWH}=使能脉冲宽度=450 ns(最小)

t_{DSW}=数据启动时间=195 ns（最小）

t_H=数据保持时间=10 ns(最小)

t_{AS}=在E信号之前，RS和R/W的启动时间=140 ns(最小)

t_{AH}=在E之后，RS和R/W的保持时间=10 ns(最小)

图 12-4　LCD 的写操作时序图(E 时序行电平从高到低)

12.1.5　LCD 数据表

在 LCD 中,数据可以被存放在任何存储单元。下面给出了地址单元以及如何访问它们。

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	1	A	A	A	A	A	A	A

其中 AAAAAAA 为 0000000～0100111 时,代表第 1 行,而 AAAAAAA 为 1000000～1100111 时,代表第 2 行。如表 12-3 所示。

表 12-3 LCD 地址

	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
第一行(最小)	1	0	0	0	0	0	0	0
第一行(最大)	1	0	1	0	0	1	1	1
第二行(最小)	1	1	0	0	0	0	0	0
第二行(最大)	1	1	1	0	0	1	1	1

对于40字符宽的LCD,上限地址范围可达到0100111,而对于20字符宽度的LCD,上限地址最高可以达到010011(十进制的19等于二进制的10011)。注意到,上限范围0100111(二进制)=39(十进制),这对应于40×2的LCD的地址0到地址39。

从上面的讨论中,可以得到各种型号LCD的光标位置的地址。图12-5是常见LCD的光标地址。所有的地址都是十六进制的。表12-4提供了LCD命令和指令的详细列表。(表12-2摘自这个表)。

16×2 LCD	80	81	82	83	84	85	86…… 8F
	C0	C1	C2	C3	C4	C5	C6……CF
20×1 LCD	80	81	82	83	……93		
20×2 LCD	80	81	82	83	……93		
	C0	C1	C2	C3	……D3		
20×4 LCD	80	81	82	83	……93		
	C0	C1	C2	C3	……D3		
	94	95	96	97	……A7		
	D4	D5	D6	D7	……E7		
40×2 LCD	80	81	82	83	……A7		
	C0	C1	C2	C3	……E7		

注意:所有数据都是十六进制形式的。

图 12-5 一些 LCD 的光标地址

表 12-4 LCD 指令列表

指令	RS R/W DB7—DB0	说明	执行时间(最大值)
Clear Display	0 0 0 0 0 0 0 0 0 1	清除整个显示,在地址计数器中将DDRAM地址设置为0	1.64 ms
Return Home	0 0 0 0 0 0 0 0 1 -	将DD RAM地址0设置为地址计数器。将显示移到原始位置。DD RAM的内容保持不变	1.64 ms
Entry Mode Set	0 0 0 0 0 0 0 1 1/D S	设置光标移动的方向,并指定显示的移动方向。这些操作在数据读写过程中进行	40 μs

（续）

指 令	RS R/W DB7—DB0	说 明	执行时间（最大值）
Display On/ Off Control	0 0 0 0 0 0 1 D C B	设置整个显示的开关状态(D)及光标的开关状态(C)，以及光标位置字符的闪烁(B)	40 μs
Cursor 或 Display Shift	0 0 0 0 0 1 S/C R/L -	移动光标和移位显示，而不改变DD RAM的内容	40 μs
Function Set	0 0 0 0 1 DL N F -	设置接口数据长度(DL)、显示的行数(L)和字符字体(F)	40 μs
Set CG RAM Address	0 0 0 1 AGC	设置CG RAM地址。CG RAM数据的发送和接收均在该设置之后	40 μs
Set DD RAM Address	0 0 1 ADD	设置DD RAM地址。DD RAM数据的发送和接收均在该设置之后	40 μs
Read Busy Flag & Address	0 1 BF AC	读busy标志位(BF)表明正在执行内部操作，并读地址计数器的内容	40 μs
Write Data CG 或 DD RAM	1 0 写数据	将数据写入DD或CG RAM	40 μs
Read Data CG 或 DD RAM	1 1 读数据	从DD或CG RAM读出数据	40 μs

注意：

(1)执行时间是当 F_{cp} 或 F_{osc} 为250 kHz时的最大时间。

(2)执行时间随频率的改变而改变。

(例如，当 F_{cp} 或 F_{osc} 为270 kHz时，40μs×250/270=37μs。)

(3)缩写标注：

DD RAM	显示数据RAM		
CG RAM	字符生成器RAM		
ACC	CG RAM地址		
ADD	DD RAM地址，和光标地址相对应		
AC	用于DD和CG RAM地址的地址计数器		
I/D=1	增量	I/D=0	减量
S=1	伴随显示移位		
S/C=1	显示移位	S/C=0	光标移动
R/L=1	移动到右端	R/L=0	移动到左端
DL=1	8位。DL=0：4位		
N=1	1行。N=0：1行		
F=1	5×10点，	F=0：	5×7点
BF=1	内部操作	BF=0	可接受指令

Optrex 是最大的 LCD 制造商之一。可以从它们的网站 http://www.optrex.com 获得数据表。

LCD 可以从下面的网站购买：http://www.digikey.com；http://www.jameco.com；http://www.elexp.com；http://www.bgmicro.com。

12.1.6 使用 TBLRD 指令向 LCD 发送信息

程序 12-3 说明了如何使用 TBLRD 指令向 LCD 发送数据和指令。

关于 LCD 的 PIC18 C 语言版本程序，请参阅程序 12-1C 和 12-2C。

程序 12-3

```
;Program 12-3: Using TableRead
;PORTD = D0-D7, RB0 = RS, RB1 = R/W, RB2 = E pins
LCD_DATA    EQU PORTD         ;LCD data pins RD0-RD7
LCD_CTRL    EQU PORTB         ;LCD control pins
RS          EQU RB0           ;RS pin of LCD
RW          EQU RB1           ;R/W pin of LCD
EN          EQU RB2           ;E pin of LCD
            CLRF   TRISD      ;PORTD = Output
            CLRF   TRISB      ;PORTB = Output
            BCF    LCD_CTRL,EN ;enable idle low
            CALL   LDELAY     ;long delay (250 ms) for power-up
            MOVLW  upper(MYCOM)
            MOVWF  TBLPTRU
            MOVLW  high(MYCOM)
            MOVWF  TBLPTRH
            MOVLW  low(MYCOM)
            MOVWF  TBLPTRL
C1          TBLRD*+
            MOVF TABLAT,W     ;give it to WREG
            IORLW 0x0         ;Is it the end of command?
            BZ     SEND_DAT   ;if yes then go to display data
            CALL   COMNWRT    ;call command subroutine
            CALL   DELAY      ;give LCD some time
            BRA    C1
SEND_DAT    MOVLW  upper(MYDATA)
            MOVWF  TBLPTRU
            MOVLW  high(MYDATA)
            MOVWF  TBLPTRH
            MOVLW  low(MYDATA)
            MOVWF  TBLPTRL
DT1         TBLRD*+
            MOVF TABLAT,W            ;give it to WREG
            IORLW  0x0        ;Is it the end of data string?
            BZ     OVER              ;if yes then exit
            CALL   DATAWRT           ;call DATA subroutine
            CALL   DELAY             ;give LCD some time
            BRA    DT1
OVER        BRA    OVER              ;stay here
```

```
COMNWRT                               ;send command to LCD
          MOVWF   LCD_DATA            ;copy WREG to LCD DATA pin
          BCF     LCD_CTRL,RS         ;RS = 0 for command
          BCF     LCD_CTRL,RW         ;R/W = 0 for write
          BSF     LCD_CTRL,EN         ;E = 1 for high pulse
          CALL    SDELAY              ;make a wide En pulse
          BCF     LCD_CTRL,EN         ;E = 0 for H-to-L pulse
          RETURN
DATAWRT                               ;write data to LCD
          MOVWF   LCD_DATA            ;copy WREG to LCD DATA pin
          BSF     LCD_CTRL,RS         ;RS = 1 for data
          BCF     LCD_CTRL,RW         ;R/W = 0 for write
          BSF     LCD_CTRL,EN         ;E = 1 for high pulse
          CALL    SDELAY              ;make a wide En pulse
          BCF     LCD_CTRL,EN         ;E = 0 for H-to-L pulse
          RETURN
          ORG     500H
MYCOM     DB 0x38,0x0E,0x01,0x06,0x84,0;commands and null
MYDATA    DB "HELLO",0                ;data and null
;look in previous chapters for delay routines
          END
```

下面这个C18程序使用延时将字母M、D和E传送至LCD。

程序 12-1C

```
//Program 12-1C: This is the C version of Program 12-1.
#include <P18F4580.h>
#define ldata PORTD              //PORTD = LCD data pins (Fig. 12-2)
#define rs PORTBbits.RB0         //rs = PORTB.0
#define rw PORTBbits.RB1         //rw = PORTB.1
#define en PORTBbits.RB2         //en = PORTB.2

void main()
  {
    TRISD = 0;                   //both ports B and D as output
    TRISB = 0;
    en = 0;                      //enable idle low
    MSDelay(250);
    lcdcmd(0x38);                //init. LCD 2 lines, 5x7 matrix
    MSDelay(250);
    lcdcmd(0x0E);                //display on, cursor on
    MSDelay(15);
    lcdcmd(0x01);                //clear LCD
    MSDelay(15);
    lcdcmd(0x06);                //shift cursor right
    MSDelay(15);
    lcdcmd(0x86);                //line 1, position 6
    MSDelay(15);
    lcddata('M');                //display letter 'M'
    MSDelay(15);
    lcddata('D');                //display letter 'D'
```

483

```
    MSDelay(15);
    lcddata('E');          //display letter 'E'
  }

void lcdcmd(unsigned char value)
  {
    ldata = value;         //put the value on the pins
    rs = 0;
    rw = 0;
    en = 1;                //strobe the enable pin
    MSDelay(1);
    en = 0;
  }
void lcddata(unsigned char value)
  {
    ldata = value;         //put the value on the pins
    rs = 1;
    rw = 0;
    en = 1;                //strobe the enable pin
    MSDelay(1);
    en = 0;
  }

void MSDelay(unsigned int itime)
  {
    unsigned int i, j;
    for(i=0;i<itime;i++)
      for(j=0;j<135;j++);
  }
```

下面是程序 12-2 的 C 语言版本，它使用了 busy 标志位的方法。

程序 12-2C

```
//Program 12-2C. C version of Program 12-2
#include <P18F458.h>
#define ldata PORTD   //PORTD = LCD data pins (Fig. 12-2)
#define rs PORTBbits.RB0    //rs = PORTB.0
#define rw PORTBbits.RB1    //rw = PORTB.1
#define en PORTBbits.RB2    //en = PORTB.2
#define busy PORTDbits.RD7  //busy = PORTD.7

void main()
  {
    TRISD = 0;              //both ports B and D as output
     TRISB = 0;
     en = 0;                //enable idle low
     MSDelay(250);          //long delay
     lcdcmd(0x38);
     MSDelay(250);          //long delay
     lcdcmd(0x0E);
     lcdready();            //check the LCD busy flag
     lcdcmd(0x01);
     lcdready();            //check the LCD busy flag
     lcdcmd(0x06);
     lcdready();            //check the LCD busy flag
     lcdcmd(0x86);          //line 1, position 6
     lcdready();            //check the LCD busy flag
     lcddata('M');
```

```
        lcdready();             //check the LCD busy flag
        lcddata('D');
        lcdready();             //check the LCD busy flag
        lcddata('E');
    }
void lcdcmd(unsigned char value)
    {
      ldata = value;            //put the value on the pins
      rs = 0;
      rw = 0;
      en = 1;                   //strobe the enable pin
      MSDelay(1);
      en = 0;
    }
void lcddata(unsigned char value)
    {
      ldata = value;            //put the value on the pins
      rs = 1;
      rw = 0;
      en = 1;                   //strobe the enable pin
      MSDelay(1);
      en = 0;
    }
void lcdready()
  {
     TRISD = 0xFF;              //make PORTD an input
      rs = 0;
      rw = 1;
      do                        //wait here for busy flag
        {
          en = 1;               //strobe the enable pin
          MSDelay(1);
          en = 0;
        }while(busy==1);
      TRISD = 0;
    }
void MSDelay(unsigned int itime)
    {
      unsigned int i, j;
      for(i=0;i<itime;i++)
        for(j=0;j<135;j++);
    }
```


程序 12-3C

```
#include <P18F458.h>
#define ldata PORTD     //PORTD = LCD data pins (Fig. 12-2)
#define rs PORTBbits.RB0            //rs = PORTB.0
#define rw PORTBbits.RB1            //rw = PORTB.1
#define en PORTBbits.RB2            //en = PORTB.2

#pragma romdata mycom = 0x300       //command at ROM addr 0x300
far rom const char mycom[] = {0x0E,0x01,0x06,0x84};

#pragma romdata mydata = 0x320      //data at ROM addr 0x320
far rom const char mydata[] = "HELLO";

void main()
  {
```

```
     unsigned char z=0;
     TRISD = 0;                  //both ports B and D as output
     TRISB = 0;
     en = 0;                     //enable idle low
     MSDelay(250);
     lcdcmd(0x38);
     MSDelay(250);
     //send out the commands
     for(;z<4;z++)
       {
         lcdcmd(mycom[z]);
         MSDelay(15);
       }
     //send out the data
     for(z=0;z<5;z++)
       {
         lcddata(mydata[z]);
         MSDelay(15);
       }
     while(1); //infinite loop
   }
   void lcdcmd(unsigned char value)
    {
      ldata = value;              //put the value on the pins
     rs = 0;
     rw = 0;
     en = 1;                     //strobe the enable pin
     MSDelay(1);
     en = 0;
   }

void lcddata(unsigned char value)
  {
    ldata = value;              //put the value on the pins
    rs = 1;
    rw = 0;
    en = 1;                     //strobe the enable pin
    MSDelay(1);
    en = 0;
  }
void MSDelay(unsigned int itime)
  {
    unsigned int i, j;
    for(i=0;i<itime;i++)
      for(j=0;j<135;j++);
  }
```


12.1.7 复习题

1. RS引脚是LCD的一个________(输入,输出)引脚。
2. E引脚是LCD的一个________(输入,输出)引脚。

3. E引脚需要一个________(由高到低,由低到高)的脉冲来锁存LCD数据引脚上的信息。
4. 若要求LCD能够将数据引脚上的信息识别为数据,则必须将RS置为________(高电平,低电平)。
5. 请给出第1行第1个字符和第2行第1个字符的命令代码。

12.2 键盘接口

键盘和LCD是最常用的输入/输出设备,对它们有基本的了解是很有必要的。本节将首先讨论键盘的基础知识,以及按键检测和按键识别原理,然后将介绍键盘和PIC18芯片的接口。

12.2.1 键盘和PIC18的接口

在最低层,键盘是以行列式的矩阵组织而成的。CPU通过端口访问行和列。因此,使用两个8位的端口可以将一个8×8的键盘连接到微处理器。当有一个键按下时,其中一行和一列便会形成一个触点。如果没有键按下,那么行和列之间便没有任何的连接。在IBM PC键盘中,有一个微控制器用来管理键盘的硬件和软件接口。在这样的系统中,存储在微控制器ROM中的程序便会不断地扫描键盘,如果发现有键被激活,便把它的信息传送到主板。如例12-3所示。键盘的接口编程需要两个步骤:(1)按键检测,(2)键识别。PIC18有两种方法进行按键检测:中断法和扫描法。在PIC18中,PORTB-Change中断可以用来执行中断按键检测。接下来将详细地介绍中断法。

例12-1 如图12-6所示,识别下面按键的行和列:RB3-RB0=1110(行),RB7-RB4=1011(列)。

解:

从图12-6中可以看出,按键的行为RB0,而列为RB6,所以键值为2号。

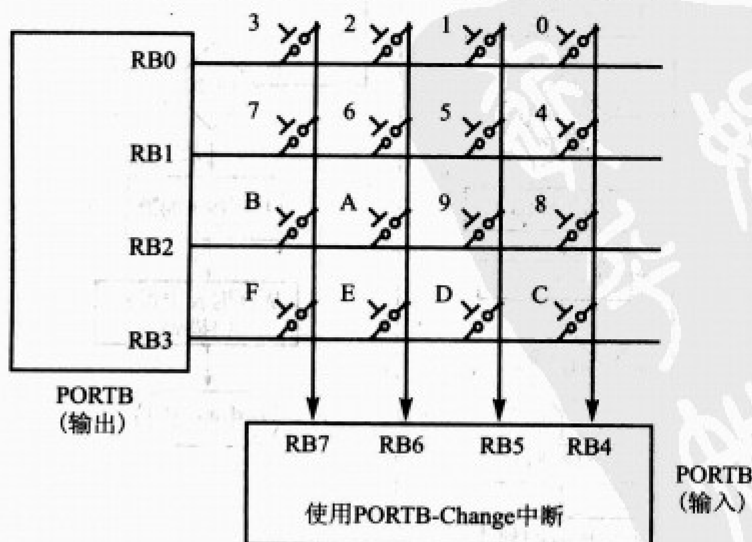


图12-6 矩阵键盘与端口的连接

用中断方法进行按键检测

图12-6是一个连接到PORTB的4×4矩阵键盘。键盘的行连接到PORTB Low (RB3~

[487] RB0)，而键盘的列连接到 PORTB. High(RB7～RB4)，即 PORTB-Change 中断。正如在 11 章讨论的，RB7～RB4 的任何改变都会导致一个中断，表明有键按下。研究程序 12-4，可以发现它由下面 4 个步骤构成。

(1) 确保按下的键被释放。所有行的输出为 0，而不断地读取和检测列的值，直到所有的列都为高电平。当所有的列都是高电平时，程序便会在等待一小段时间后进入等待按键按下的第(2)步。

(2) 检测是否有键被按下。因为按键的列连接到 PORTB-Change 中断，所以如果有任何键被按下都会导致中断，然后微控制器将执行 ISR。ISR 必须做两件工作：(a)保证按键检测不会在尖峰脉冲噪声的影响下出现错误；(b)等待 20 ms，以防止同一个按键被解释成多个按键。图 12-8 是键盘的防抖图。

(3) 为了检测按键属于哪一行，微控制器每次将一行接地，并同时读取各列状态。如果它检测到所有列都为高电平，这说明按键不属于该行；因此，它继续将下一行接地，并不断重复上面的工作，直到它找到按键所属的行为止。一旦找到按键所属的行，它将设置查询表的起始地址，可用于保持扫描码(或者 ASCII 值)，并进入下一个步骤以识别按键。

(4) 为了识别按键，微控制器每次需要将一个列位传送到进位标志位，以检验它是否为低电平。如果发现了低电平，那么微控制器便从查询表中将该键的 ASCII 代码取出；否则，它便将指针加 1 指向查询表的下一个单元。图 12-7 是该过程的流程图。

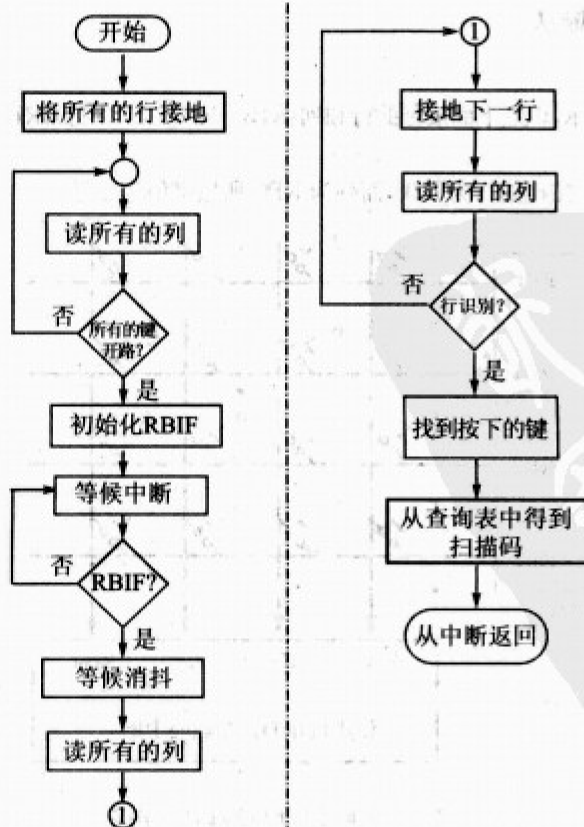


图 12-7 程序 12-4 的流程图

将程序 12-4 中的查表法做适当修改后，可用于任何不大于 8×4 的矩阵。图 12-7 提供了扫描识别按键的程序 12-4 的流程图。

研究程序 12-4，注意到程序使用中断来检测按键，而 ISR 的工作便是识别按键。程序 12-4C 是程序 12-4 的 C18 版本。在汇编程序(12-4)中，字符存放在 PORTD，而在 C18 版本中，字符被送到串行端口并在显示器上显示。

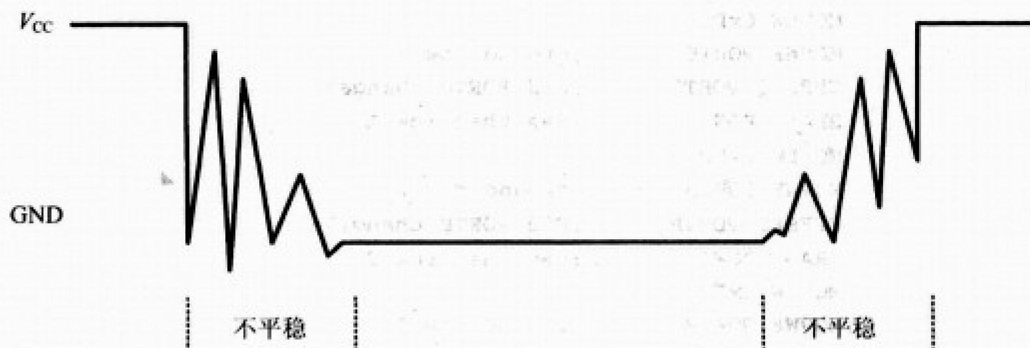


图 12-8 键盘的防抖动

程序 12-4 本程序使用 PORTB 等待按键按下，然后将字符置于端口 PORTD。在程序中假设：RB3～RB0 连接到行，而 RB7～RB4 连接到列

```
D15mH       EQU   D'100'      ;15 ms delay high byte of value
D15mL       EQU   D'255'      ;low byte of value
COL         EQU   0x08        ;holds the column found
DR15mH      EQU   0x09        ;registers for 15 ms delay
DR15mL      EQU   0x0A        ;
;-----------------------------------------
            ORG   0x000000
RESET_ISR   GOTO  MAIN        ;jump over interrupt table
            ORG   0x000008
HI_ISR      BTFSC INTCON,RBIF ;Was it a PORTB change?
            BRA   RBIF_ISR    ;yes then go to ISR
            RETFIE            ;else return
;------------program for initialization
MAIN        CLRF  TRISD       ;make PORTD output port
            BCF   INTCON2,RBPU;enable PORTB pull-up resistors
            MOVLW 0xF0        ;make PORTB high input ports
            MOVWF TRISB       ;make PORTB low output ports
            MOVWF PORTB       ;ground all rows
KEYOPEN     CPFSEQ PORTB      ;are all keys open
            GOTO  KEYOPEN     ;wait until keypad ready
            MOVLW upper(KCODE0)
            MOVWF TBLPTRU     ;load upper byte of TBLPTR
            MOVLW high(KCODE0)
            MOVWF TBLPTRH     ;load high byte of TBLPTR
            BSF   INTCON,RBIE ;enable PORTB change interrupt
            BSF   INTCON,GIE  ;enable all interrupts globally
LOOP        GOTO  LOOP        ;wait for key press
```

488
～
490

```
;------------key identification ISR
RBIF_ISR    CALL   DELAY          ;wait for debounce
            MOVFF PORTB,COL       ;get the column of key press
            MOVLW 0xFE
            MOVWF PORTB           ;ground row 0
            CPFSEQ PORTB          ;Did PORTB change?
            BRA    ROW0           ;yes then row 0
            MOVLW 0xFC
            MOVWF PORTB           ;ground row 1
            CPFSEQ PORTB          ;Did PORTB change?
            BRA    ROW1           ;yes then row 1
            MOVLW 0xFB
            MOVWF PORTB           ;ground row 2
            CPFSEQ PORTB          ;Did PORTB change?
            BRA    ROW2           ;yes then row 2
            MOVLW 0xF7
            MOVWF PORTB           ;ground row 3
            CPFSEQ PORTB          ;Did PORTB change?
            BRA    ROW3           ;yes then row 3
            GOTO   BAD_RBIF       ;no then key press too short
ROW0        MOVLW low(KCODE0)     ;set TBLPTR = start of row 0
            BRA    FIND           ;find the column
ROW1        MOVLW low(KCODE1)     ;set TBLPTR = start of row 1
            BRA    FIND           ;find the column
ROW2        MOVLW low(KCODE2)     ;set TBLPTR = start of row 2
            BRA    FIND           ;find the column
ROW3        MOVLW low(KCODE3)     ;set TBLPTR = start of row 3
FIND        MOVWF TBLPTRL         ;load low byte of TBLPTR
            MOVLW 0xF0
            XORWF COL             ;invert high nibble
            SWAPF COL,F           ;bring to low nibble
AGAIN       RRCF   COL            ;rotate to find column
            BC     MATCH          ;column found, get the ASCII code
            INCF   TBLPTRL        ;else point to next col. address
            BRA    AGAIN          ;keep searching
MATCH       TBLRD*+               ;get ASCII code from table
            MOVFF TABLAT,PORTD    ;display pressed key on PORTD
WAIT        MOVLW 0xF0
            MOVWF PORTB           ;reset PORTB
            CPFSEQ PORTB          ;Did PORTB change?
            BRA    WAIT           ;yes then wait for key release
            BCF    INTCON,RBIF    ;clear PORTB, change flag
            RETFIE                ;return and wait for key press
BAD_RBIF    MOVLW 0x00            ;return null
            GOTO   WAIT           ;wait for key release
;-------------delay
DELAY:      MOVLW D15mH           ;high byte of delay
            MOVWF DR15mH          ;store in register
D2:         MOVLW D15mL           ;low byte of delay
```

```
         MOVWF DR15mL      ;store in register
D1:      DECF  DR15mL.F    ;stay until DR15mL becomes 0
         BNZ   D1
         DECF  DR15mH,F    ;loop until all DR15m = 0x0000
         BNZ   D2
         RETURN
;-------------key scancode look-up table
         ORG   300H
KCODE0:  DB    '0','1','2','3'    ;ROW 0
KCODE1:  DB    '4','5','6','7'    ;ROW 1
KCODE2:  DB    '8','9','A','B'    ;ROW 2
KCODE3:  DB    'C','D','E','F'    ;ROW 3
         END
```

程序 12-4C 介绍了 PIC18 键盘接口的 C 编程。

程序 12-4C　这个程序读取键盘并将结果发送到串行端口。在程序中假设：RB0～RB3 连接到行，而 RB4～RB7 连接到列。串行端口设置为波特率 9600(10MHz XTAL)，8 位模式，1 个停止位

```
#include <p18f458.h>
void SerTX(unsigned char x);
void RBIF_ISR(void);
void MSDelay(unsigned int millisecs);
unsigned char keypad[4][4] = {'0','1','2','3',
                              '4','5','6','7',
                              '8','9','A','B',
                              'C','D','E','F'};

#pragma code My_HiPrio_Int =0x0008 //high-priority interrupt
void My_HiPrio_Int (void)
{
  _asm
    GOTO chk_isr
  _endasm
}
#pragma code

#pragma interrupt chk_isr     //which ISR
void chk_isr (void)
{
  if (INTCONbits.RBIF==1)     //RBIF caused interrupt?
  RBIF_ISR( );                //yes go to RBIF_ISR
}
#pragma code

void main()
{
  TRISD=0;                      //make PORTD output port
  INTCON2bits.RBPU=0;           //enable PORTB pull-up resistors
  TRISB=0xF0;             //PORTB low as output and high as input
  PORTB=0xF0;                   //clear PORTB low
  while(PORTB!=0xF0);           //wait until key not pressed
  TXSTA=0x20;                   //choose low baud rate, 8-bit
  SPBRG=15;                     //9600 baud rate, XTAL = 10 MHz
  TXSTAbits.TXEN=1;             //enable transmit
  RCSTAbits.SPEN=1;             //enable serial port
  INTCONbits.RBIE=1;            //enable PORTB interrupt on change
  INTCONbits.GIE=1;             //enable interrupts globally
  while(1);                     //wait until key press
}
void RBIF_ISR(void)             //finds the key pressed
{
```

492

```
    unsigned char temp,COL=0,ROW=4;
    MSDelay(15);
    temp = PORTB;                    //get column
    temp ^= 0xF0;                    //invert high nibble
    if(!temp) return;                //if false alarm return
    while(temp<<=1) COL++;           //find the column
    PORTB = 0xFE;                    //ground row 0
    if(PORTB != 0xFE)                //Did high nibble change?
      ROW = 0;                       //yes then row 0
    else {                           //try next row
      PORTB = 0xFD;                  //ground row 1
      if(PORTB != 0xFD)              //Did high nibble change?
        ROW = 1;                     //yes then row 1
      else {                         //try next row
        PORTB = 0xFB;                //ground row 2
        if(PORTB != 0xFB)            //Did high nibble change?
          ROW = 2;                   //yes then row 2
        else {                       //try last row
          PORTB = 0xF7;              //ground row 3
            if(PORTB != 0xF7)        //Did high nibble change?
              ROW = 3;               //yes then row 3
        }
      }
    }
    if(ROW<4)                        //Did we find a valid row?
      SerTX(keypad[ROW][COL]);       //then send character
    while(PORTB!=0xF0) PORTB=0xF0;      //wait for release
    INTCONbits.RBIF=0;               //reset flag
  }
  void SerTX(unsigned char x)        //sends character
  {
    while(PIR1bits.TXIF!=1);         //wait until ready
    TXREG=x;                         //send character out serial port
  }

 void MSDelay(unsigned int millisecs)
 {
   unsigned int i, j;
   for(i=0;i<millisecs;i++)
     for(j=0;j<135;j++);
 }
```

493

12.2.2　使用扫描法进行按键检测

按键检测的另一种方法是扫描法。在这种检测方法中，微控制器需要通过向输出锁存器输出0将所有行接地，然后读取各列的状态。各行数据为1111表明没有键按下，因此微控制器将继续检测，直到检测到有键按下。若各列的位中有一个位为0，则说明有键按下。检测到有键按下后，微控制器将进行按键识别。首先，微控制器将第1行接地，然后读取各列的状态。若各列值均为1，则说明在这一行中没有键按下，微控制器接着检测下一行。同样地，它会将下一行接地，然后读取各列的状态，检验各列的数值是否为0。这个过程将不断地重复，直到键所在的行被识别出来。在按下的键所在的行被检测出之后，微控制器便会继续找出按键所在的列。这种识别应该是很容易的，因为微控制器知道所访问的行和列。图12-9是该扫描法的流程图。该程序的实现留给读者。

有些IC芯片（如美国国家半导体公司的MM74C923），将键盘扫描和解码功能均集成于芯片内部。有些芯片使用计数器和逻辑门（没有微控制器）来实现本章的基础概念。

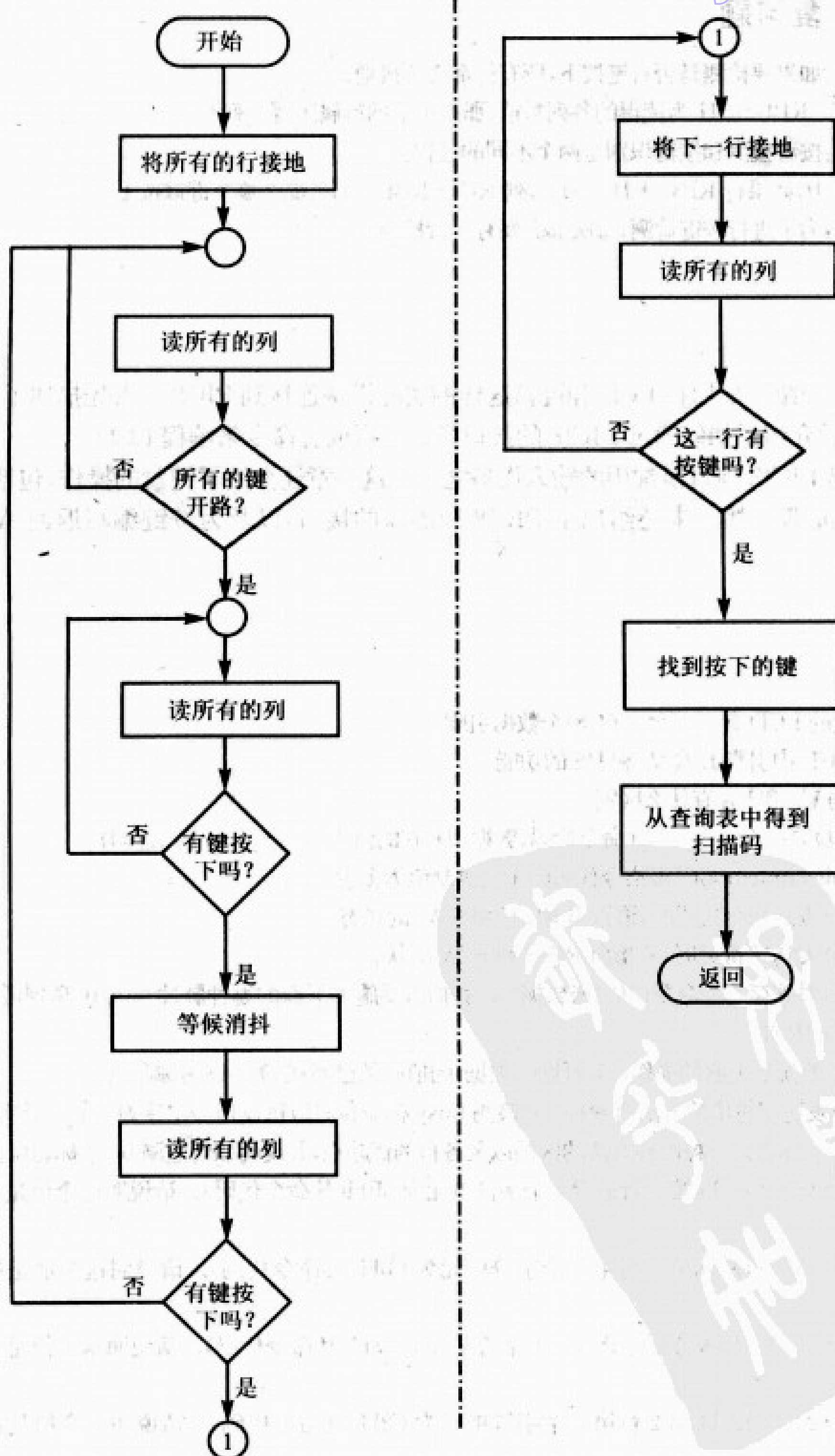


图 12-9 用于按键检测的扫描法的流程图

12.2.3 复习题

1. 判断对错:如果要检测是否有键按下,所有行都必须接地。
2. 如果 RB7—RB4=0111 为读得的各列数值,那么按下的键属于哪一列?
3. 判断对错:按键检测和按键识别是两个不同的过程。
4. 在图 12-6 中,如果行 RB3—RB0=1110,列 RB7—RB4=1110,那么哪个键被按下?
5. 判断对错:为了进行按键检测,每次都需要有一行接地。

494~495

小结

这一章介绍了如何将 LCD 和键盘这样的实际设备连接到 PIC18。首先描述了 LCD 的操作模式,然后介绍了如何通过 PIC18 的接口发送数据或者命令来编程 LCD。

键盘是 PIC18 项目最常用的输入设备之一。这一章还介绍了键盘的操作,包括按键检测和按键识别原理。接下来还给出了 PIC18 和键盘的接口,以及为按键编写返回 ASCII 码的 PIC18 程序。

习题

1. 本节讨论的 LCD 有________(4,8)个数据引脚。
2. 请描述 LCD 中引脚 E、R/W 和 RS 的功能。
3. LCD 中的 V_{CC} 和 V_{EE} 有什么区别?
4. Clear LCD 是一个________(命令代码,数据项),它的值为________(十六进制)。
5. 对于 display on,cursor on,指令代码的十六进制值为多少?
6. 当向 LCD 发送指令代码时,请指出 RS、E 和 R/W 的状态。
7. 当向 LCD 发送字符 Z 时,请指出 RS、E 和 R/W 的状态。
8. 如果 LCD 要锁存一个命令代码(或数据),E 引脚需要施加下面的哪种脉冲? (a)由高到低的脉冲,(b)由低到高的脉冲
9. 判断对错:要实现上面的操作,指令代码(数据)的值必须已经在 D0~D7 引脚上。
10. 向 LCD 发送字符串的方法有两种:(1)检测 busy 标志位;(2)每发送一个字符进行一次延时,而无需检测 busy 标志。解释两种方法的区别以及各自的优缺点,并说明如何监测 busy 标志位。
11. 对于一个 16×2 LCD,第 1 行最后一个字符单元为 8FH(其命令代码)。请说明这个值是如何计算出来的。
12. 对于一个 16×2 LCD,第 2 行第一个字符单元为 C0H(其命令代码)。请说明这个值是如何计算出来的。
13. 对于一个 20×2 LCD,第 2 行最后一个字符单元为 93H(其命令代码)。请说明这个值是如何计算出来的。
14. 对于一个 20×2 LCD,第 2 行第三个字符单元为 C2H(其命令代码)。请说明这个值是如何计算出来的。
15. 对于一个 40×2 LCD,第 1 行最后一个字符单元为 A7H(其命令代码)。请说明这个值是如何计算出来的。

496

16. 对于一个 40×2 LCD,第 2 行最后一个字符单元为 E7H(其命令代码)。请说明这个值是如何计算出来的。
17. 写出 20×2 LCD 的第 1 行第 10 个单元的命令代码的值(十六进制),并写出计算过程。
18. 写出 40×2 LCD 的第 2 行第 20 个单元的命令代码的值(十六进制),并写出计算过程。
19. 假设 RC4=RS,RC5=R/W,RC6=E,重新编写 COMNWRT 子例程。
20. 重复第 19 题,编制数据写入子例程。通过检测 busy 位的方法将字符串 Hello 传送至 LCD,并使用指令 TBLRD。
21. 在读键盘矩阵的列值时,如果没有键按下,那么读得的列值都为 ________(1,0)。
22. 在程序 12-4 中,为了检测按键,下面哪个操作会被执行?
    (a)PORTB—Change 中断　(b)每次将一行接地
23. 在图 12-6 中,为了检测按键,下面哪一项需要接地?
    (a)所有行　(b)每次一行　(c)(a)和(b)都选
24. 在图 12-6 中,如果 RB7—RB4=0111,RB3—RB0=1110,那么哪个键按下?
25. 指出使用 IC 芯片代替微控制器进行键盘扫描和解码的优缺点。
26. 请给出第 25 题答案的最折衷方案。

复习题答案

12.1 节

1. 输入　2. 输入　3. 由高到低　4. 高电平　5. 80H 和 C0H

12.2 节

1. 正确。　2. 第 3 列　3. 正确。　4. 0　5. 正确。

第 13 章 ADC、DAC 和传感器接口

学习目标：

- ❑ PIC18 芯片的 ADC(模数转换器)
- ❑ PIC18 的温度传感器接口
- ❑ ADC 的数据采掘过程
- ❑ 选择 ADC 芯片的考虑因素
- ❑ PIC18 的 ADC 的汇编编程和 C 编程
- ❑ DAC(数模转换器)芯片的基本操作
- ❑ PIC18 的 DAC 接口
- ❑ DAC 芯片的编程，在示波器上产生正弦波
- ❑ PIC18 的 DAC 的汇编编程和 C 编程
- ❑ 高精度 IC 温度传感器的功能
- ❑ 信号调整及其在数据采掘中的作用

本章将探讨更多的实际设备，如 ADC(模数转换器)、DAC(数模转换器)和传感器。本章还会介绍 PIC18 和这些设备的接口。13.1 节介绍 ADC 芯片。13.2 节学习 PIC18 芯片的 ADC 部件编程。13.3 节讨论 ADC 芯片的特性。13.4 节学习传感器接口和信号调整问题。

13.1 ADC 特性

本节将探讨 PIC18 芯片的 ADC 编程。首先将介绍 ADC 的一些通用特性，然后介绍如何使用 C 语言和汇编语言对 PIC18 的 ADC 部件编程。

13.1.1 ADC 设备

模数转换器是使用最广泛的数据捕获设备之一。数字计算机使用二进制(离散)数值，但是物理世界是模拟(连续)的，例如每天都会碰到的一些物理量：温度、(气体或者液体的)压强、湿度和速度。要将物理量转换成电信号(电压、电流)，需要使用一个叫作转换器的设备。该转换器也叫作传感器。用于温度、速度、压强、光还有其他自然量的传感器都会产生电压(电流)的输出量。因此，需要一个模数转换器把模拟信号转换成微控制器可读、可处理的数字信号。如图 13-1 和图 13-2 所示。

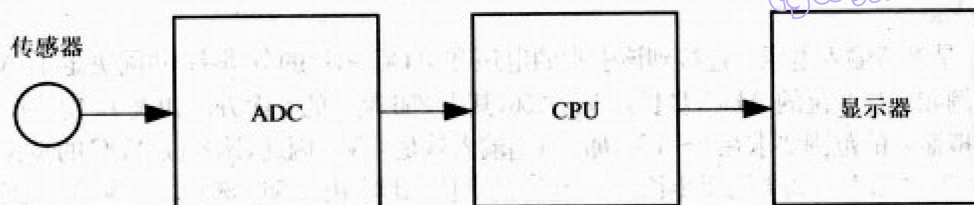


图 13-1 微控制器通过 ADC 连接到传感器

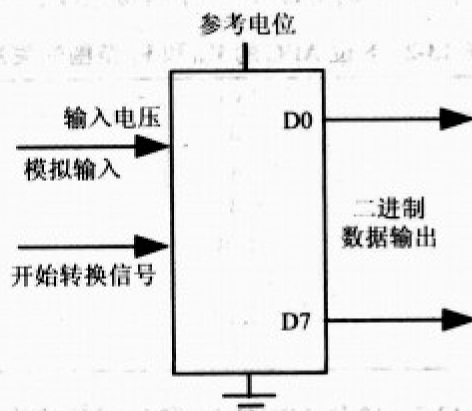


图 13-2 8 位 ADC 的方框图

500

ADC 的主要特性有以下几个。

1. 分辨率

ADC 有 n 位的分辨率，其中 n 可以是 8、10、12、16，甚至是 24。分辨率越高，则 ADC 的步长越小。步长是指 ADC 可识别的最小变化量。表 13-1 给出了一些常见的 ADC 分辨率。尽管一块 ADC 芯片的分辨率在其设计的时候已经固定，是无法改变的，但是可以通过 V_{ref} 来控制步长。下面将会加以讨论。

表 13-1 ADC 分辨率和步长(参考电位 V_{ref}=5V)

n 位	步　数	步长(mV)
8	256	5/256 = 19.53
10	1 024	5/1024 = 4.88
12	4 096	5/4096 = 1.2
16	65 536	5/65 536 = 0.076

注意：Vcc=5 V。

步长(分辨率)是指 ADC 可识别的最小变化量。

2. 转换时间

除了分辨率之外，转换时间也是 ADC 的一个重要的评价因素。转换时间是指 ADC 用来把模拟输入转换成数字(二进制)信号所需的时间。转换时间的决定因素包括 ADC 的时钟源，用于数据捕获的方法，以及 ADC 芯片的生产工艺(使用 MOS 还是 TTL 技术)。

3. V_{ref}

V_{ref}是参考输入电压。连接到该引脚的电压和ADC芯片的分辨率,共同决定了ADC的步长。例如一片8位的ADC,步长是$V_{ref}/256$,其中256是2的8次方。如表13-1所示。如果它的模拟输入的范围要求在0~4 V,那么V_{ref}输入就是4 V。因此,该8位ADC的步长就是4 V/256=15.62 mV。如果需要步长为10 mV的8位ADC,由2.56/256=10 mV,就得到V_{ref}=2.56 V。如果一个10位的ADC,V_{ref}=5 V,那么它的步长就是4.88 mV。如表13-1所示。表13-2和表13-3分别给出了8位和10位ADC步长与V_{ref}的关系。在有些应用中还需要差动的参考电压,$V_{ref}=V_{ref}(+)-V_{ref}(-)$。通常,$V_{ref}(-)$引脚接地,$V_{ref}(+)$引脚用作$V_{ref}$。

表13-2 8位ADC的V_{ref}和V_{in}范围的关系

V_{ref}(V)	V_{in}(V)	步长(mV)
5.00	0~5	5/256 = 19.53
4.0	0~4	4/256 = 15.62
3.0	0~3	3/256 = 11.71
2.56	0~2.56	2.56/256 = 10
2.0	0~2	2/256 = 7.81
1.28	0~1.28	1.28/256 = 5
1	0~1	1/256 = 3.90

501

表13-3 10位ADC的V_{ref}和V_{in}范围的关系

V_{ref}(V)	V_{in}(V)	步长(mV)
5.00	0~5	5/1024 = 4.88
4.096	0~4.096	4.096/1024 = 4
3.0	0~3	3/1024 = 2.93
2.56	0~2.56	2.56/1024 = 2.5
2.048	0~2.048	2.048/1024 = 2
1.28	0~1.28	1.28/1024 = 1.25
1.024	0~1.024	1.024/1024 = 1

4. 数字信号输出

在8位ADC中,有8位的数字信号输出——D0~D7;而在10位ADC中,有10位的数字信号输出——D0~D9。为了计算输出电压,可以使用下面的公式:

$$D_{out}=V_{in}/\text{步长}$$

其中,D_{out}是数字输出电压(十进制形式),V_{in}是模拟输入电压,步长(分辨率)是最小的变化量。对于8位ADC,步长=$V_{ref}/256$。请参阅例13-1。ADC芯片可以每次输出一位数(串行),也可以每次并行输出一组数。下面将讨论这两种芯片。

例13-1 假设一个8位的ADC,V_{ref}=2.56 V。计算下面模拟输入在D0~D7上的输出:(a)1.7 V (b)2.1 V。

解:

因为步长是2.56/256=10 mV,所以得到:

(a)D_{out}=1.7 V/10 mV=170(十进制形式),因此,D0~D7的二进制数输出是10101010。

(b)D_{out}=2.1 V/10 mV=210(十进制形式),因此,D0~D7 的二进制数输出是 11010010。

5. ADC 的并行和串行输出

ADC 芯片有并行和串行两种。对于并行 ADC,有 8 个或者更多的引脚来输出二进制数据,而对于串行 ADC,则只有一个输出引脚。也就是说,在串行 ADC 里,有一个并-串移位寄存器,使得每次只输出一位二进制数。8 位 ADC 的 D0~D7 数据引脚在 ADC 芯片与 CPU 间之间提供了 8 位并行的数据通道。对于 16 位并行 ADC 芯片,则需要 16 位的数据通道。为了节省引脚,很多 12 位和 16 位的 ADC 使用 D0~D7 引脚分别输出二进制数据的高字节和低字节。近年来,在许多应用中,空间是一个很重要的考虑因素,所以把大量引脚用在数据输出上是不切实际的。因此,如串行 ADC 类的串行设备应用得更多。尽管串行 ADC 使用更少的引脚和更小的封装,占用更小的印制电路板空间,然而却需要更多的 CPU 时间来转换 ADC 数据,而不是像并行 ADC 那样简单地读取数据。ADC848 是一种 8 引脚输出的并行 ADC,而 MAX1112 是一种单数据输出引脚(D_{out})的串行 ADC。图 13-3 和图 13-4 分别画出了 ADC848 和 MAX1112 的方框图。

502

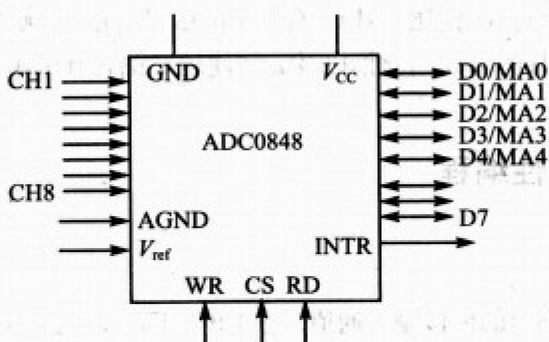


图 13-3 ADC0848 并行 ADC 的方框图

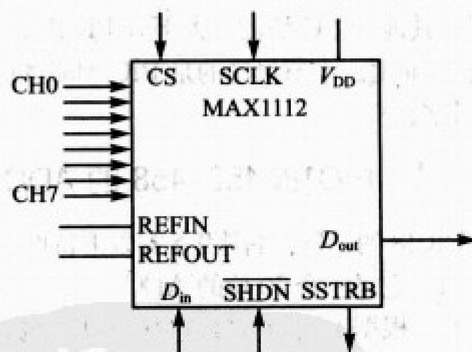


图 13-4 MAX1112 串行 ADC 的方框图

503

6. 模拟输入通道

很多数据捕获应用需要不只一个 ADC。因此,可以看到一块 ADC 芯片上有 2、4、8 甚至 16 个通道。在 ADC848 和 MAX1112 上也可以看到有多路的模拟输入。在这类芯片上,有 8 个模拟输入通道,用来捕获多个物理量,如温度、压强、热度等。PIC18 微控制器芯片有 5~15 个数量不等的 ADC 通道,这取决于微控制器的型号。PIC18 的 ADC 特性将在下一节讨论。

7. 转换开始和转换结束信号

由于使用的是多路模拟输入通道和单数字输出寄存器,所以必须使用转换开始(SC)和转换结束(EOC)信号。当 SC 被触发时,ADC 开始把 V_{in} 输入的模拟量转换成 n 位的数字信号。转换所需的时间跟前面提到的转换方法有关。当数据转换完成时,转换结束信号会通知 CPU 数据已转换好并等待读取。

从讨论中可以得到 ADC 芯片转换数据的步骤如下。

(1) 选择一个通道。

(2) 触发转换开始(SC)信号来启动模拟输入的转换。

(3) 一直监视转换结束(EOC)信号。

(4) 当EOC被触发时，读取ADC芯片的输出数据。

13.1.2 复习题

1. 说出影响步长的两个因素。
2. ADC0848是一个________位的转换器。
3. 判断对错：ADC0848有8位 D_{out} 引脚，而MAX1112只有一个 D_{out} 引脚。
4. 指出下面ADC芯片的模拟输入通道数目。
   (a)ADC0848　　(b)MAX1112
5. 请指出8位ADC的步长，假设 $V_{ref}=1.28$ V。
6. 根据第5题的条件，计算下面模拟输入在D0～D7上的输出：(a)0.7 V，(b)1 V。

504

13.2 PIC18的ADC编程

因为ADC广泛应用在数据捕获中，所以近年来，越来越多的微控制器片内就内置ADC，就像内置定时器和USART一样。片内ADC避免了对外部ADC的需求，因此可把更多的引脚留给其他的I/O活动。大部分PIC18芯片带有8通道的ADC，有些PIC18芯片的ADC则多达16通道。本节讨论的是PIC18452/458芯片的ADC特性，以及如何使用C语言和汇编语言对其编程。

13.2.1 PIC18F452/458的ADC特性编程

PIC18的ADC外围设备有以下特性。

(1) 它是一个10位的ADC。

(2) 根据不同的型号，它有5～15个数量不等的模拟输入通道。在PIC18452/458中，引脚PORTA的RA0～RA7是用作8位模拟通道的。如图13-5a和图13-5b所示。

(3) 转换后的输出二进制数存放在两个特殊寄存器ADRESL(A/D结果的低位)和ADRESH(A/D结果的高位)中。

(4) ADRESL寄存器提供的是16位的数据宽度，而ADC输出数据只有10位，因此其中的6位是未使用的。读者可以选择该6位的未使用位是在高6位还是在低6位。

(5) 可以选用PIC18芯片自身的电压源 $V_{DD}(V_{CC})$ 来作为 V_{ref}，也可以选用外部电压源作为 V_{ref}。

(6) 数据转换的时间由OSC引脚的晶振频率决定。当PIC18的晶振的最高频率为40 MHz时，转换时间应不小于1.6 ms。

(7) 使用 $V_{ref}(+)$ 和 $V_{ref}(-)$，根据 $V_{ref}=V_{ref}(+)-V_{ref}(-)$，它就可以得到不同的差动的 V_{ref} 电压。

上面的很多特性都可以使用ADCON0(A/D控制寄存器0)和ADCON1(A/D控制寄存器1)来编程。请继续往下看。

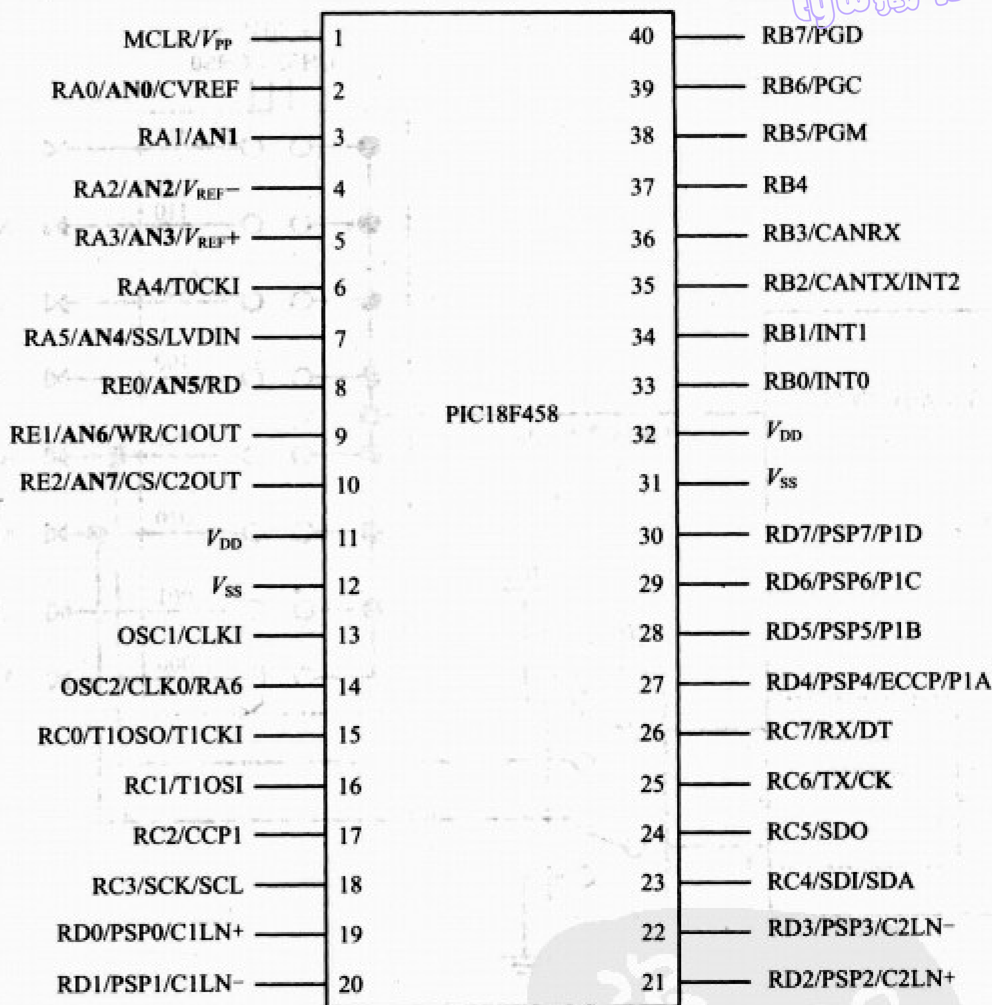


图 13-5a　PIC18F458 的 ADC 通道引脚(图中粗体字部分)

505 ~ 506

13.2.2　ADCON0 寄存器

ADCON0 寄存器用于设置转换时间和选择模拟输入通道。图 13-6 画出了 ADCON0 寄存器。为了减少 PIC18 的功耗,在微控制器启动的时候,ADC 功能是关闭的。使用 ADCON0 寄存器的 ADON 位可以打开 ADC,如图 13-6 所示。另一个重要的位是 GO/DONE。使用该位来启动转换和监视转换是否结束。注意,对于 ADCCON0,并不是所有型号的芯片都有 8 个模拟输入通道。转换时间由 ADCS 位来设置。ADCS1 和 ADCS0 都位于 ADCON0 寄存器,而 ADCS2 是 ADCON1 寄存器的一部分。下面将会加以介绍。

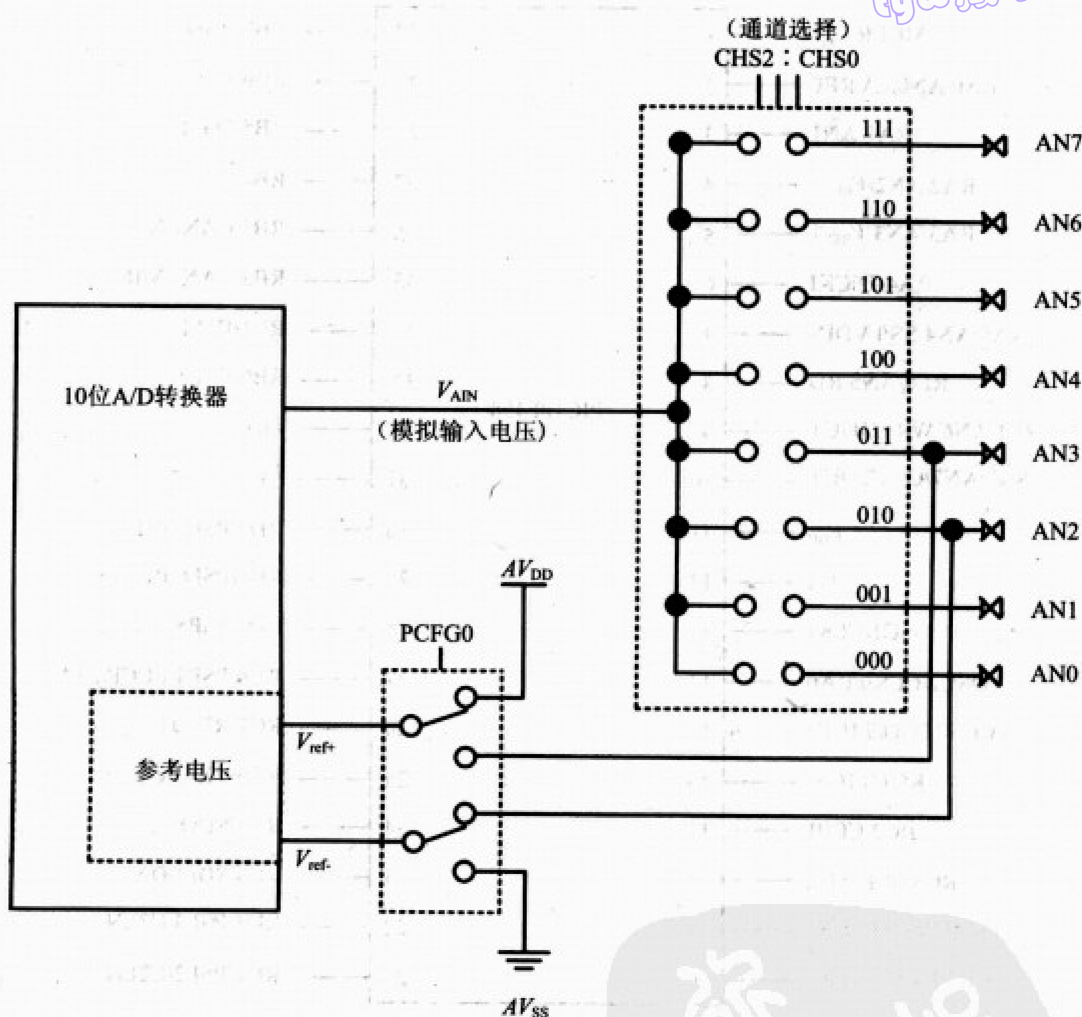


图 13-5b PIC18 的 ADC 通道和参考电压选择

ADCS1	ADCS0	CHS2	CHS1	CHS0	GO/DONE	——	ADON

ADCS2（属于 ADCON1）	ADCS1	ADCS0	转换时钟源
0	0	0	$F_{osc}/2$
0	0	1	$F_{osc}/8$
0	1	0	$F_{osc}/32$
0	1	1	内部 RC 作为时钟源
1	0	0	$F_{osc}/4$
1	0	1	$F_{osc}/16$
1	1	0	$F_{osc}/64$
1	1	1	内部 RC 作为时钟源

图 13-6 ADCON0 寄存器(A/D控制寄存器 0)

CHS2	CHS1	CHS0	通道选择
0	0	0	CHAN0(AN0)
0	0	1	CHAN1(AN1)
0	1	0	CHAN2(AN2)
0	1	1	CHAN3(AN3)
1	0	0	CHAN4(AN4)
1	0	1	CHAN5(AN5)不适用于 28 引脚的 PIC18
1	1	0	CHAN6(AN6)不适用于 28 引脚的 PIC18
1	1	1	CHAN7(AN7)不适用于 28 引脚的 PIC18

GO/DONE A/D 转换状态位

1=A/D 转换正在进行。该位用于启动转换。也就是说当转换完成时，该位会变为低电平，表明转换结束

0=A/D 转换已经完成，寄存器的数字信息可用

ADON A/D 使能位

0=PIC18 的 A/D 部分关闭，不消耗功率。这是默认值。当不需要使用 ADC 时，可以保持它的关闭状态

1=A/D 功能被启用

507

图 13-6 (续)

13.2.3 ADCON1 寄存器

PIC18 的 ADC 的另一个主要的寄存器是 ADCON1。ADCON1 寄存器用于选择 V_{ref} 电压，如下面图 13-7 所示。当 A/D 转换完成时，结果放在寄存器 ADRESL(A/D 结果的低字节)和 ADRESH(A/D 结果的高字节)中。由于只用到了 16 位中的 10 位，所以可以使用 ADCON1 寄存器的 ADFM 位来选择是左对齐还是右对齐方式。如图 13.8 所示。

ADFM	ADCS2	——	——	PCFG3	PCFG2	PCFG1	PCFG0

ADFM A/D 结果格式选择位

1=右对齐：10 位的结果分别放在 **ADRESL** 寄存器和 **ADRESH** 的低 2 位。也就是说，**ADRESH** 的高 6 位全部置 0

2=左对齐：10 位的结果分别放在 **ADRESH** 寄存器和 **ADRESL** 的高 2 位。也就是说，**ADRESL** 的低 6 位全部置 0

ADCS2 A/D 时钟选择位 2。该位同 ADCON0 寄存器的 ADCS1 位以及 ADCS0 位，共同决定了 ADC 的转换时钟。ADCS2 的默认值是 0，即通过设置 ADCON0 的 ADCS0 位和 ADCS1 位的值来提供 $F_{osc}/2$、$F_{osc}/8$ 和 $F_{osc}/32$ 的时钟。请参考 ADCON0 寄存器

PCFG:A/D 端口配置控制位

PCFGs	AN7	AN6	AN5	AN4	AN3	AN2	AN1	AN0	$V_{ref}+$	$V_{ref}-$	C/R
0000	A	A	A	A	A	A	A	A	V_{DD}	V_{SS}	8/0

图 13-7 ADCON1(A/D 控制寄存器 1)

0001	A	A	A	A	$V_{ref}+$	A	A	A	AN3	V_{SS}	7/1
0010	D	D	D	A	A	A	A	A	V_{DD}	V_{SS}	5/0
0011	D	D	D	A	$V_{ref}+$	A	A	A	AN3	V_{SS}	4/1
0100	D	D	D	D	A	D	A	A	V_{DD}	V_{SS}	3/0
0101	D	D	D	D	$V_{ref}+$	D	A	A	AN3	V_{SS}	2/1
011x	D	D	D	D	D	D	D	D	—	—	0/0
1000	A	A	A	A	$V_{ref}+$	$V_{ref}-$	A	A	AN3	AN2	6/2
1001	D	D	A	A	A	A	A	A	V_{DD}	V_{SS}	6/0
1010	D	D	A	A	$V_{ref}+$	A	A	A	AN3	V_{SS}	5/1
1011	D	D	A	A	$V_{ref}+$	$V_{ref}-$	A	A	AN3	AN2	4/2
1100	D	D	D	A	$V_{ref}+$	$V_{ref}-$	A	A	AN3	AN2	3/2
1101	D	D	D	A	$V_{ref}+$	$V_{ref}-$	A	A	AN3	AN2	2/2
1110	D	D	D	D	D	D	D	A	V_{DD}	V_{SS}	1/0
1111	D	D	D	D	$V_{ref}+$	$V_{ref}-$	D	A	AN3	AN2	1/2

A=模拟输入，D=数字 I/O

C/R=模拟输入通道/A/D 引脚参考电压

默认值是 0000，提供 8 个模拟输入通道，使用 PIC18 的 V_{DD} 作为参考电压

508

图 13-7 （续）

A/D 通道的端口配置由 PCFG(A/D 端口控制)位决定。例如，对于 PIC18452/458 芯片，有多达 8 个的模拟输入通道，不过并不是所有的应用都需要使用这么多的 ADC 输入。PORTA 的 RA0～RA3 和 RA5 引脚，以及 PORTE 的 RE0～RE2 引脚都可以用于模拟输入通道。当 PCFG=1001 时，使用 PORTA 的所有引脚作为数字 I/O。PCFG 的默认值是 0000，允许使用 8 个模拟输入引脚。这时，$V_{ref}=V_{DD}$，即 PIC18 芯片自身的电压源。很多时候，还需要用到其他电源作为 V_{ref}。AN3 引脚可以用作 V_{ref} 的外部电压源。例如，置 PCFG=0101，允许使用 2 个模拟输入，AN3=V_{ref}，而其他 5 个引脚用作数字 I/O。这时，PIC18 的 V_{SS}(Gnd)引脚作为 V_{ref}(—)。请看例 13-2 和例 13-3。

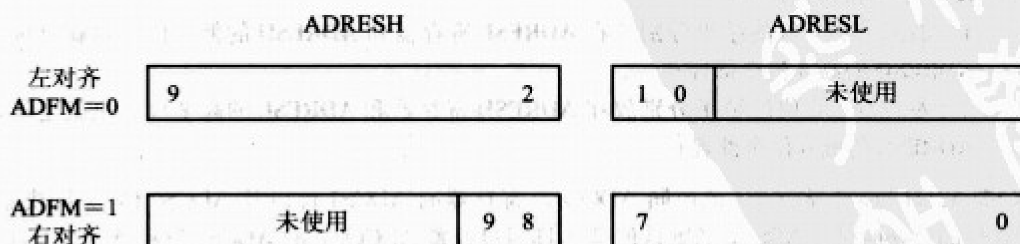


图 13-8 ADFM 位和 ADRESx 寄存器

例 13-2 对于一个基于 PIC18 的系统，设 $V_{ref}=V_{DD}=5$ V。(a)计算步长，(b)如果需要 3 个通道，计算 ADCON1 的值。假设 ADRESH：ADRESL 寄存器是右对齐的。

解：

(a) 步长=5/1024=4.8 mV

(b) ADCON1=1x0001000，因为 100 表示 3 个模拟输入通道。x=ADCS2，由转换速度决定。

例 13-3 对于一个基于 PIC18 的系统，设 $V_{ref}=2.56$ V。(a) 计算步长，(b) 如果需要 3 个通道，计算 ADCON1 的值。假设 ADRESH：ADRESL 寄存器是右对齐的。

解：

(a) 步长=2.56/1024=2.5 mV

(b) ADCON1=1x000011，因为 0011 表示 3 个模拟输入通道。x=ADCS2，由转换速度决定。

509

13.2.4 计算 A/D 转换时间

使用 ADCON0 和 ADCON1 寄存器的 ADCS(A/D 时钟源)置位，可以设定 A/D 转换时间。转换时间由 T_{ad}决定。T_{ad}指每位数据的转换时间。为了计算 T_{ad}，需要从 $F_{osc}/2$、$F_{osc}/4$、$F_{osc}/8$、$F_{osc}/16$、$F_{osc}/32$ 和 $F_{osc}/64$ 中选择转换时钟源，其中 F_{osc} 是 PIC18 的晶振频率。对于 PIC18，转换时间是 T_{ad}的 12 倍。注意，T_{ad}不能小于 1.6 ms。请看例 13-4 和例 13-5 来理清这些概念。

除了外部晶振 F_{osc}，还可以使用内部 RC 晶振作为转换时钟源。这时，T_{ad}通常是 4 μs～6 μs，转换时间就是 12×6 μs=72 μs。

另一个需要注意的时间是捕获时间(T_{acq})。在选择好 A/D 通道以后，取样-保持电容需要一些充电时间以达通道上的输入电压水平。只有经过这些捕获时间，A/D 转换才能开始。虽然很多因素(如 V_{dd}和温度)会影响 T_{acq}的大小，但是可以使用 15 μs 作为典型值。对于一些新型的 PIC18，可以通过对它内部的 ADCON2 寄存器编程来精确控制 T_{acq}。对于 PIC18F452/458，只有 ADCON0 和 ADCON1 寄存器。请看例 13-6。

例 13-4 PIC18 连接有 10 MHz 晶振。对 ADCON0 和 ADCON1 寄存器中 ADCS 的所有选项，计算对应的转换时间。

解：

ADCON0 和 ADCON1 的时钟源有下面几种情况。

(a) 对于 $F_{osc}/2$，有 10 MHz/2=5 MHz。

T_{ad}=1/5 MHz=200 ns。因为小于 1.6 μs，所以是无效的。

(b) 对于 $F_{osc}/4$，有 10 MHz/4=2.5 MHz。

T_{ad}=1/2.5 MHz=400 ns。因为小于 1.6 μs，所以是无效的。

(c) 对于 $F_{osc}/8$，有 10 MHz/8=1.25 MHz。

T_{ad}=1/1.25 MHz=800 ns。因为小于 1.6 μs，所以是无效的。

(d) 对于 $F_{osc}/16$，有 10 MHz/16=625 kHz。

T_{ad}=1/625 kHz=1.6 μs。转换时间=12×1.6 μs=19.2 μs。

(e) 对于 $F_{osc}/32$，有 10 MHz/32=312.5 kHz。

T_{ad}=1/312.5 kHz=3.2 μs。转换时间=12×3.2 μs=38.4 μs。

(f) 对于 $F_{osc}/64$，有 10 MHz/64=156.25 kHz。

T_{ad}=1/156.25 kHz=6.4 μs。转换时间=12×6.4 μs=76.8 μs。

注意，对于 $F_{osc}/4$、$F_{osc}/16$ 和 $F_{osc}/64$，除了用到 ADCON0 寄存器的两个 ADSC 位外，还用到 ADCON1 寄存器的 ADSC2 位。

510

例 13-5 PIC18 连接有 4 MHz 晶振。若只使用 ADCON0 寄存器中的 ADCS 位，则计算转换时间。

解：

对于 ADCON0 寄存器，可用的转换时钟源有下面几种情况。

(a) 对于 $F_{osc}/2$，有 4 MHz/2=2 MHz。
T_{ad}=1/2 MHz=500 ns。因为小于 1.6 μs，所以是无效的。

(b) 对于 $F_{osc}/8$，有 4MHz/8=500 kHz。
T_{ad}=1/500 kHz=2 μs。转换时间=12×2 μs=24 μs。

(c) 对于 $F_{osc}/32$，有 4 MHz/32=125 kHz。
T_{ad}=1/125 kHz=8 μs。转换时间=12×8 μs=96 μs。

例 13-6 找出下面选项对应的 ADCON0 和 ADCON1 的值：(a)AN0 通道作为模拟输入，(b)$V_{ref}+=V_{dd}$，$V_{ref}-=V_{ss}$，(c)$F_{osc}/64$，(d)A/D结果是右对齐的，(e)A/D模块打开。

解：

根据图 13-6，可以得到 ADCON0=10000x1。当 x=0 时，就有 10000001。

根据图 13-6，可以得到 ADCON0=11xx1110。当 x=0 时，就有 11001110。

13.2.5 使用查询法对 A/D 转换器编程

PIC18 的 A/D 转换器编程，要遵循下面的步骤。

(1) 打开 PIC18 的 ADC 模块。因为在上电时为了节省功耗，ADC 是关闭的。可以使用指令 `BSF ADCON0,ADON` 打开。

(2) 选择引脚作为 ADC 通道的输入引脚。可以用指令 `BSF TRISA, x` 和 `BSF TRISE, x`，其中 x 是通道的编号。

(3) 选择参考电压和 A/D 输入通道。可以使用 ADCON0 和 ADCON1 寄存器。

(4) 选择转换速度。可以使用 ADCON0 和 ADCON1。

(5) 等待要求的捕捉时间到达。

(6) 触发转换开始位 GO/GONE。

(7) 用查询法监视转换结束(GO/GONE)位，等待转换完成。

(8) 当 GO/DONE 位变成低电平时，读取 ADRESL 和 ADRESH 寄存器，获取数字输出信号。

511

(9) 返回第(5)步。

13.2.6 PIC18F458 ADC 的汇编语言编程

汇编语言程序 13-1 体现了上面介绍的 ADC 转换步骤。它的 C 语言版本程序在程序 13-1C 中给出。

程序 13-1 该程序从ADC的通道0(RA0)读取数据，然后把结果显示在PORTC和PORTD。它每隔1/4 s工作一次

```
Program 13-1: This program gets data from channel 0 (RA0)
of ADC and displays the result on PORTC and PORTD. This is
done every quarter of second.
;Program 13-1
        ORG   0000H
        CLRF   TRISC            ;make PORTC an output
        CLRF   TRISD            ;make PORTD an output
        BSF    TRISA,0 ;make RA0 an input for analog input
        MOVLW  0x81             ;Fosc/64, channel 0, A/D is on
        MOVWF  ADCON0
        MOVLW  0xCE  ;right justified, Fosc/64, AN0 = analog
        MOVWF  ADCON1
OVER    CALL   DELAY ;wait for Tacq (sample and hold time)
        BSF   ADCON0,GO         ;start conversion
BACK    BTFSC ADCON0,DONE       ;keep polling end-of-conversion

        BRA   BACK              ;wait for end of conversion
        MOVFF  ADRESL,PORTC     ;give the low byte to PORTC
        MOVFF  ADRESH,PORTD     ;give the high byte to PORTD
        CALL   QSEC_DELAY
        BRA   OVER              ;keep repeating it
        END
```

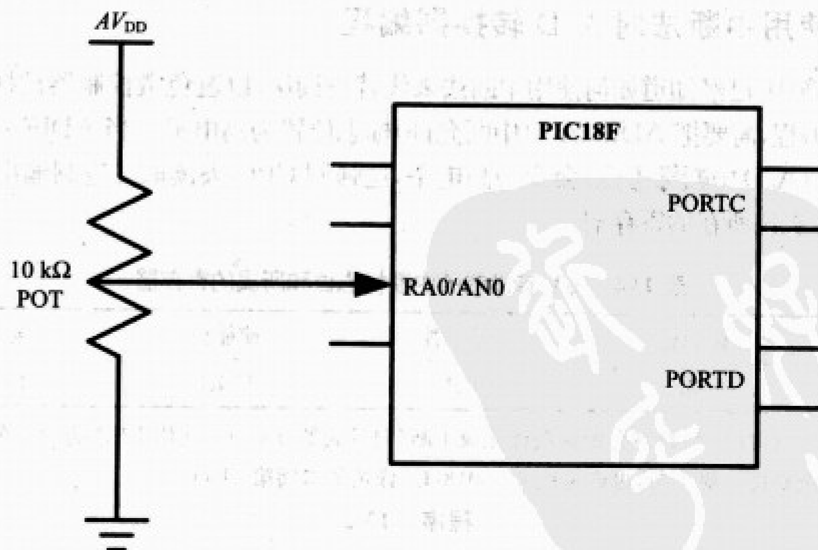


图 13-9 程序 13-1 的 A/D 连接

13.2.7 PIC18F458 A/D 的 C 语言编程

程序 13-1C 是 ADC 转换程序 13-1 的 C 语言版本。

程序 13-1C 该程序从ADC的通道0(RA0)读取数据，然后把结果显示在PORTC和PORTD。它每隔1/4 s工作一次。它是程序13-1C的C语言版本

```
Program 13-1C: This program gets data from channel 0 (RA0)
of ADC and displays the result on PORTC and PORTD. This is
```

```
done every quarter of second. This is the C version of
Program 13-1.
//Program 13-1C

void main(void)
  {
      TRISC=0;              //make PORTC output port
      TRISD=0;              //make PORTD output port
      TRISAbits.TRISA0=0;   //RA0 = INPUT for analog input
      ADCON0 = 0x81;    //Fosc/64, channel 0, A/D is on
      ADCON1 = 0xCE;        //right justified, Fosc/64,
                            //AN0 = analog
      while(1)
      {
       DELAY(1); //give A/D channel time to sample
       ADCON0bits.GO = 1;   //start converting
       while(ADCON0bits.DONE == 1);
       PORTC=ADRESL;        //display low byte on PORTC
       PORTD=ADRESH;        //display high byte on PORTD
       DELAY(250);          //wait for one quarter of a
                            //second before trying again
      }
  }
```

13.2.8 使用中断法对A/D转换器编程

在第11章中,已经知道如何使用中断法来代替查询法,以避免微控制器被捆绑。要用中断法对A/D编程,需要把ADIE(A/D中断允许)标志位置为高电平。当ADIE=1时,在转换完成后,ADIF(A/D中断标志位)会变为高电平,这强制CPU去读取二进制输出。表13-4给出了这两个标志位所在的寄存器。

表13-4 A/D转换器的中断标志位和所属的寄存器

中断	标志位	寄存器	使能位	寄存器
ADIF(ADC)	ADIF	PIR1	ADIE	PIE1

513

注意:在上电复位时,A/D被分配为高优先级中断(向量表的0008)。使用IPR1寄存器的ADIP位来为它分配低优先级中断,也就是地址00018H。详情请参阅第11章。

程序 13-2

```
;Program 13-2
      ORG   0000H
      GOTO  MAIN        ;bypass interrupt vector table
;--on default all interrupts go to to address 00008
      ORG   0008H       ;interrupt vector table
      BTFSS PIR1,ADIF   ;Did we get here due to A/D int?
      RETFIE            ;No. Then return to main
      GOTO   AD_ISR     ;Yes. Then go INT0 ISR
;--the main program for initialization
      ORG    00100H
MAIN  CLRF   TRISC      ;make PORTC an output
      CLRF   TRISD      ;make PORTD an output
```

```
        BSF    TRISA,0  ;make RA0 an input pin for analog input
        MOVLW  0x81         ;Fosc/64, channel 0, A/D is on
        MOVWF  ADCON0
        MOVLW  0xCE   ;right justified, Fosc/64, AN0 = analog
        MOVWF  ADCON1
        BCF PIR1,ADIF      ;clear ADIF for the first round
        BSF PIE1,ADIE      ;enable A/D interrupt
        BSF INTCON,PEIE    ;enable peripheral interrupts
        BSF INTCON,GIE     ;enable interrupts globally
OVER    CALL DELAY         ;wait for Tacq (sample and hold time)
        BSF   ADCON0,GO    ;start conversion
        BRA OVER           ;stay in this loop forever
;-----A/D Converter ISR
AD_ISR
        ORG 200H
        MOVFF ADRESL,PORTC     ;give the low byte to PORTC
        MOVFF ADRESH,PORTD     ;give the high byte to PORTD
        CALL QSEC_DELAY
        BCF PIR1,ADIF      ;clear ADIF interrupt flag bit
        RETFIE
        END
```

程序 13-2C

```
//Program 13-2C (This is the C version of Program 13-2)

#include <PIC18F458.h>

#pragma code My_HiPrio_Int=0x0008 //high-priority interrupt
void My_HiPrio_Int (void)
{
  chk_isr();
}
#pragma code                  //end high-priority interrupt
#pragma interrupt chk_isr     //Which interrupt?
void chk_isr (void)
{
  if (PIR1bits.ADIF==1)       //A/D caused interrupt?
  AD_ISR( );                  //Yes. Execute INT0 program
}
void main(void)
{
  TRISC=0;                    //make PORTC output port
  TRISD=0;                    //make PORTD output port
  TRISAbits.TRISA0=0;         //RA0 = INPUT for analog input
  ADCON0 = 0x81;              //Fosc/64, channel 0, A/D is on
  ADCON1 = 0xCE; //right justified, Fosc/64, AN0 = analog
  PIR1bits.ADIF=0;            //clear A/D interrupt flag
  PIE1bits.ADIE=1;            //enable A/D interrupt
  INTCONbits.PEIE=1;          //enable peripheral interrupts
  INTCONbits.GIE=1;           //enable all interrupts globally
  while(1)              //keep looping until interrupt comes
  {
    DELAY(1);
    ADCON0bits.GO = 1;        //start converting
```

```
    }
}
//----------A/D ISR
void AD_ISR(void)
   {
     PORTC=ADRESL;          //display low byte on PORTC
     PORTD=ADRESH;          //display high byte on
     DELAY(250);            //wait for one quarter of a
                            //second before trying again
     PIR1bits.ADIF=0;       //clear A/D interrupt flag
   }
```

13.2.9 复习题

1. 说出影响 PIC18 的 A/D 步长的主要因素。
2. PIC18 的 A/D 是一个______位的转换器。
3. 判断对错：PIC18 的 A/D 有 D_{out} 引脚。
4. 判断对错：PIC18 的 A/D 是片外模块。
5. 如果 V_{ref}=1.024 V，请计算 PIC18 的 ADC 的步长。
6. 对于第 5 题的条件，计算下面输入的 D0~D9 输出：(a)0.7 V，(b)1 V。
7. 对于下面的 ADCON0 寄存器选项，请给出可用的模拟输入通道数量。
   (a)PCFG=0100　(b)PCFG=1001
8. 判断对错：转换时间等于 $12\times T_{ad}$。
9. 最小允许 T_{ad}是______ μs。
10. 哪个位是用于查询转换结束的？

515

13.3 DAC 接口

本节将介绍 DAC(数模转换器)和 PIC18 的接口以及如何使用 DAC 在示波器上产生一个正弦波。

13.3.1 数模转换器(DAC)

数模变送器(DAC)是广泛地用于把数字脉冲转换成模拟信号的一种设备。在本节中，将讨论 DAC 和 PIC18 的接口基础。

回想在数字电路课程里面学过的两种产生 DAC 的方法：二进制加权和倒 T 形电阻网络。为了得到更高的精准度，大部分的集成电路 DAC[包括本节用到的 MC1408(DAC0808)]都使用的是倒 T 形网络方法。评价 DAC 的第一个标准就是分辨率，它是二进制输入位数的函数。通常的 DAC 都是 8 位、10 位和 12 位的。输入数据的位数决定了 DAC 的精度，因为输出的模拟量的电平等级等于 2^n，其中 n 是输入的位数。因此，一个 8 位输入的 DAC，如 DAC0808，可以提供 256 个离散的输出电平等级，如图 13-10 所示。相似地，一个 12 位 DAC 能提供 4906 个离散的电平等级。此外，还有 16 位的 DAC，不过它们要贵得多。

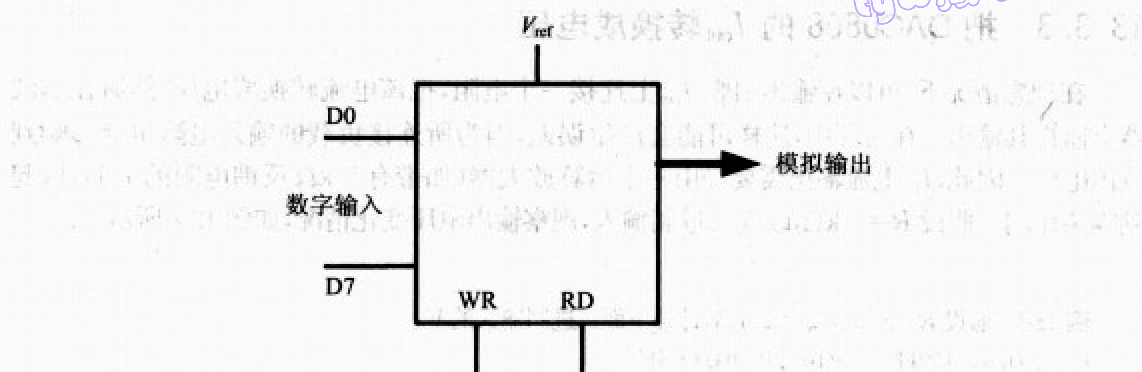


图 13-10 DAC 方框图

13.3.2 MC1408 DAC(或 DAC0808)

MC1408(或 DAC0808)把数字输入转换成电流(I_{out}),经由一个连接到 I_{out} 引脚的电阻,把结果转换成电压。I_{out} 引脚提供的总电流是 DAC0808 输入端 D0~D7 的二进制数和参考电压的一个函数,如下所示:

$$I_{out}=I_{ref}\left(\frac{D7}{2}+\frac{D6}{4}+\frac{D5}{8}+\frac{D4}{16}+\frac{D3}{32}+\frac{D2}{64}+\frac{D1}{128}+\frac{D0}{256}\right)$$

其中,D0 是输入的最低位,D7 是最高位,I_{ref} 是施加到引脚 14 的输入电流。I_{ref} 电流通常是 0.2 mA。图 13-11 画出了使用标准 5 V 电源来产生参考电流的原理图(设 I_{ref}=2 mA)。假定 I_{ref}=2 mA,如果 DAC 的所有输入端都为高电平,那么输出的最大电流就是 1.99 mA(请读者自己证明)。 516

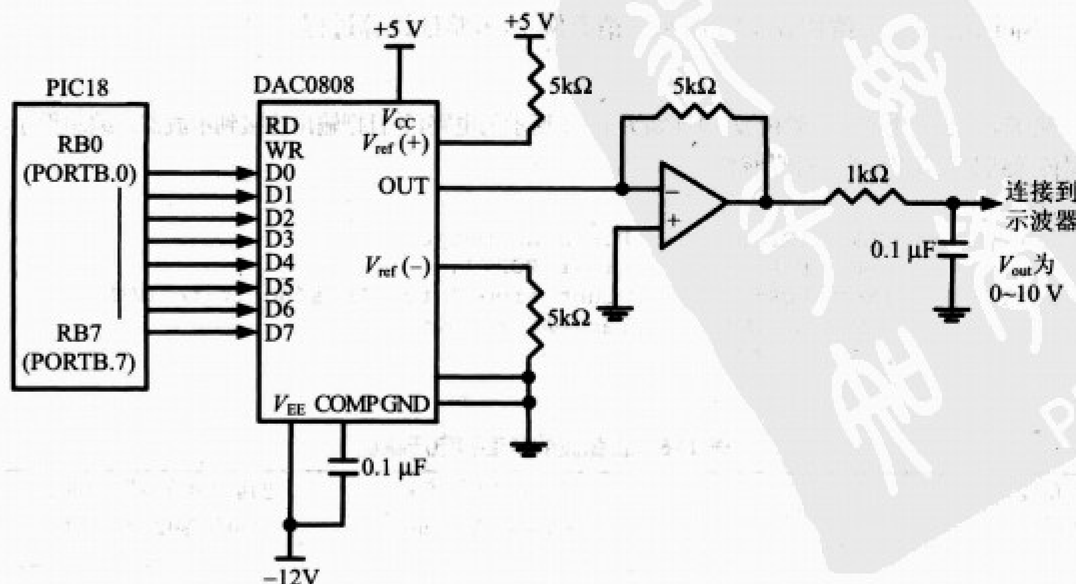


图 13-11 PIC18 连接到 DAC0808

13.3.3 把DAC0808的I_{out}转换成电压

在理想情况下,可以在输出引脚I_{out}上连接一个电阻,把该电流转换成电压,然后在示波器上监视其输出。在实际中,这样可能会产生误差,因为所连接负载的输入电阻也会影响到输出电压。因此,I_{ref}电流输出需要经由一个运算放大器(如带有5 kΩ反馈电阻的741),以起到隔离作用。假设R=5 kΩ,改变二进制输入,观察输出电压变化情况,如例13-7所示。

例13-7 假设R=5 kΩ,I_{ref}=2 mA,计算下面二进制输入的V_{out}:

(a) 10011001(99H) (b) 11001000(C8H)

解:

(a) I_{out}=2 mA(153/256)=1.195 mA,而V_{out}=1.195 mA×5 kΩ=5.975 V

(b) I_{out}=2 mA(200/256)=1.562 mA,而V_{out}=1.562 mA×5 kΩ=7.8125 V

13.3.4 产生正弦波

例13-8介绍了如何生成阶梯波。要生成正弦波,首先需要一张含有正弦函数从0°到360°对应幅值的表。从0°到360°,正弦函数的值在−0.1到+0.1之间变化。因此,表中的值就是表征θ的正弦值的电压幅度的整数值。这种方法保证了PIC18微控制器的DAC输出只能是整数。表13-5列出了角度、正弦值、电平值,还有每个角度(间隔为30°)对应的电平的整数值。为了得到表13-5,假设DAC输出电压最大值是10 V(如图13-11的设计)。当DAC输入端的所有位都为高电平时,DAC的输出就达到最大值。因此,要达到最大的10 V输出,可以使用下面的方程:

517

$$V_{out}=5\text{ V}+(5\times\sin\theta)$$

不同角度的V_{out}值如表13-5所示。请看例13-9,验证计算过程。

例13-8 为了生成一个阶梯波,设计如图13-11所示的电路,并且把输出连接到示波器。编制程序,把数据传送到DAC,以产生一个阶梯波。

解:

```
            CLRF  TRISB          ;PORTB as output
            CLRF  PORTB          ;clear PORTB
AGAIN:      INCF  PORTB,F        ;count from 0 to FFH send it to DAC
            RCALL DELAY          ;let DAC recover
            BRA   AGAIN
```

表13-5 正弦波的角度和电压幅度

角度θ (°)	$\sin\theta$	V_{out}(电压幅度) 5 V+(5 V×$\sin\theta$)	送到DAC的值(十进制) (电压幅度×25.6)
0	0	5	128
30	0.5	7.5	192

（续）

角度 θ (°)	sinθ	V_{out}（电压幅度）5 V+(5 V×sinθ)	送到 DAC 的值（十进制）（电压幅度×25.6）
60	0.866	9.33	238
90	1.0	10	255
120	0.866	9.33	238
150	0.5	7.5	192
180	0	5	128
210	−0.5	2.5	64
240	−0.866	0.669	17
270	−1.0	0	0
300	−0.866	0.669	17
330	−0.5	2.5	64
360	0	5	128

518

例 13-9 求解下面角度对应的数字输入量：(a)30°，(b)60°。

解：

(a) V_{out}=5 V+(5 V×sinθ)=5 V+5×sin30°=5 V+5×0.5=7.5 V

DAC 输入值=7.5 V×25.6=192（十进制）

(b) V_{out}=5 V+(5V×sinθ)=5 V+5×sin60°=5 V+5×0.866=9.33 V

DAC 输入值=9.33 V×25.6=238（十进制）

为了计算出不同角度对应的 DAC 输入值，可以简单地把 V_{out} 乘以 25.60，因为共有 256 个电平且 V_{out} 最大值为 10 V。于是，256/10 V=25.6/V。为了更清楚地阐述这个概念，请看下面的程序代码。该程序通过向 DAC 连续发送数据（无限循环），生成了一个粗糙的正弦波，如图 13-12 所示。

程序 13-3

```
;Program 13-3
OVER   MOVLW  upper(TABLE)
       MOVWF  TBLPTRU
       MOVLW  high(TABLE)
       MOVWF  TBLPTRH
       MOVLW  low(TABLE)
       MOVWF  TBLPTRL
       CLRF   TRISB
AGAIN  TBLRD*
       MOVF   TABLAT,W
       XORLW  0x0
       BZ     OVER
       MOVWF  PORTB
```

```
        INCF  TBLPTRL,F
        BRA   AGAIN
        ORG   0x250
TABLE:        DB D'128',D'192',D'238',D'255',D'238',D'192'
        DB D'128',D'64',D'17',D'1',D'17',D'64',D'0'
        END

        ;to get a better looking sine wave, regenerate
        ;Table 13-5 for 2-degree angles
```

519

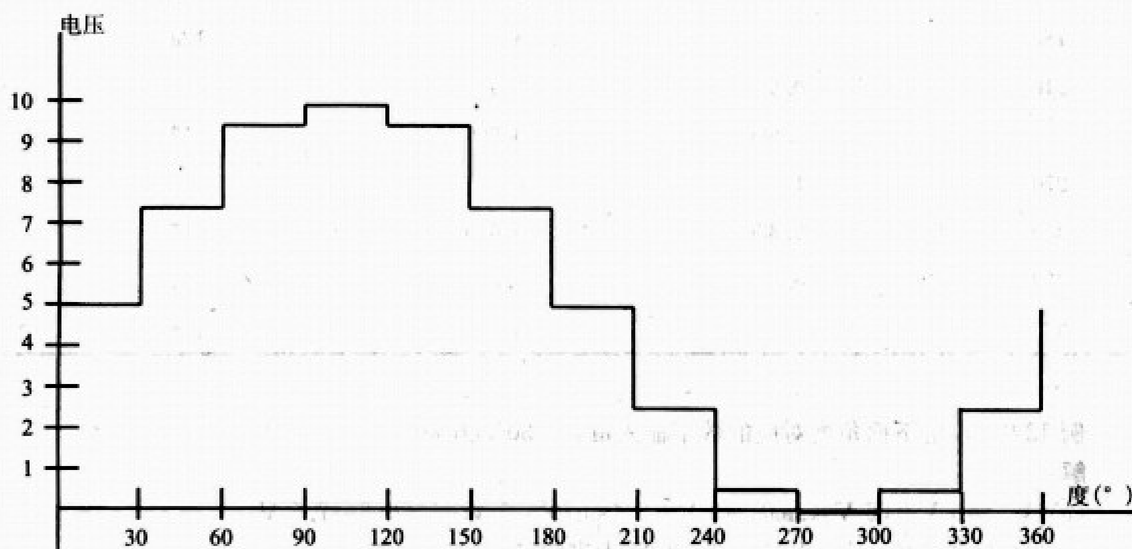


图 13-12 正弦波的角度和电平幅度

13.3.5 DAC 的 C 语言编程

程序 13-3C

```
//Program 13-3C. This is the C version of Program 13-3.
#include <p18F458.h>
rom const unsigned char WAVEVALUE[12] ={128,192,238,255,
                    238,192,128,64,
                    17,0,17,64};
void main()
  {
    unsigned char x;
    TRISB=0;
    while(1)
      {
      for(x=0;x<12;x++)
       PORTB = WAVEVALUE[x];
      }
  }
```

13.3.6 复习题

1. 对于 DAC,输入是________(数字,模拟)信号,输出是________(数字,模拟)信号。
2. 对于 ADC,输入是________(数字,模拟)信号,输出是________(数字,模拟)信号。
3. DAC0808 是一个________位的 D/A 转换器。
4. (a) DAC0808 的输出是________(电流,电压)。
   (b) 判断对错:DAC0808 的输出对于驱动电机非常理想。

520

13.4 传感器接口和信号调整

本节将介绍微控制器和传感器的接口。也将提到一些流行的温度传感器,并讨论信号调整问题。虽然本节把重点放在了温度传感器,但信号调整的基本原理同样适用于其他传感器(如光传感器和压力传感器)。

13.4.1 温度传感器

转换器把物理量(如温度、光强、流量和速度)转换成电信号。输出的形式可以是电压、电流、电阻或者是电容,这取决于传感器。例如,热敏电阻是把温度转换成电信号的传感器件。热敏电阻将温度的变化反映在了电阻值的变化上,但这种变化是非线性的,如表13-6所示。

表 13-6 热敏电阻和温度的对照

温度(℃)	Tf(kΩ)
0	29.490
25	10.000
50	3.893
75	1.700
100	0.817

由于编写非线性器件程序是很复杂的,因此很多厂家瞄准了线性温度传感器市场。简单易用又广受欢迎的线性温度传感器,包括美国国家半导体公司(National Semiconductor Corp.)的 LM34 和 LM35 系列。下面将介绍它们。

13.4.2 LM34 和 LM35 温度传感器

LM34 系列传感器是一种精密的集成电路温度传感器,它的输出电压与华氏温度成线性比例。如表 13-7 所示。由于 LM34 是内部校准的,所以不需要外来校准。每一度华氏温度对应着 10 mV 的输出。表 13-7 是 LM34 的选用指南。

LM35 系列传感器是一种精密的集成电路温度传感器,它的输出电压与摄氏温度成线性比例。由于 LM35 是内部校准的,所以不需要外来校准。每一度摄氏温度对应着 10 mV 的输出。表 13-8 是 LM35 的选用指南。(想了解更多信息,可登录 http://www.national.com。)

表 13-7 LM34 温度传感器系列选择指南

型 号	温度范围	精 度	输出规格
LM34A	−50F～+300F	+2.0 F	10mV/F
LM34	−50F～+300F	+3.0 F	10mV/F
LM34CA	−40F～+230F	+2.0 F	10mV/F

(续)

型　号	温度范围	精　度	输出规格
LM34C	−40F～+230F	+3.0 F	10mV/F
LM34D	−32F～+212F	+4.0 F	10mV/F

521

注意：温度范围使用的是华氏温度。

表 13-8　LM35 温度传感器系列选择指南

型　号	温度范围	精　度	输出规格
LM35A	−55℃～+150℃	+1.0℃	10mV/℃
LM35	−55℃～+150℃	+1.5℃	10mV/℃
LM35℃A	−40℃～+110℃	+1.0℃	10mV/℃
LM35℃	−40℃～+110℃	+1.5℃	10mV/℃
LM35D	0℃～+100℃	+2.0℃	10mV/℃

注意：温度范围使用的是摄氏温度。

13.4.3　信号调整和 PIC18 的 LM35 接口

信号调整是数据捕获中经常用到的技术。大多数传感器的输出只有电压、电流、电荷、电容和电阻中的一种形式。然而，为了向 A/D 转换器输入信号，需要把这些信号转换成电压。这个转换过程就称为信号调整。如图 13-13 所示。信号调整可以是从一个电流到电压的转换，也可以是一个信号的放大。例如，热敏电阻根据温度的变化来改变阻值；为了用于 ADC，则需要将电阻的变化转换成电压。请看一个将 LM34 连接到 PIC18F458 的模数转换器的例子。A/D 有 10 位的分辨率和最大值为 1024 的电平数，而温度每变化一度，对应于 LM34(或者是 LM35)的输出有 10 mV 的增减。如果使用 10 mV 的步长，那么 V_{out} 的最大输出值就是 10 240 mV(10.24 V)。然而，即使 LM34 的输入达到最大的 300°F，上面的值也是不允许的，因为 A/D 的有效最大输出是 3000 mV(3.00 V)。如果现在使用 2.5 mV 作为步长，那么最大的 V_{out} =1024×2.5 mV=2560 mV(2.56 V)。因此，必须将 V_{ref} 设为 2.56 V。这样得到的 A/D 二进制输出是真实温度的 4 倍(10 mV/2.5 mV=4)。将输出除以 4，就可以得到实际的温度。如表 13-9 所示。

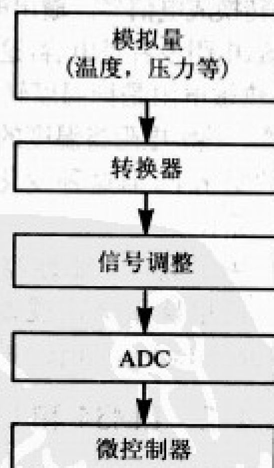


图 13-13　将模拟量转换为数据

表 13-9　温度与 PIC18 V_{out} 的对照(V_{ref}=2.56 V)

温度(F)	V_{in}(mV)	电平数	V_{out}(二进制，b9～b0)	温度(二进制)
0	0	0	00 00000000	00000000
1	10	4	00 00000100	00000100
2	20	8	00 00001000	00000010
3	30	12	00 00001100	00000011
10	100	20	00 00101000	00001010
20	200	80	00 01010000	00010100
30	300	120	00 011110000	00011110
40	400	160	00 10100000	00101000

（续）

温度(F)	V_{in}(mV)	电平数	V_{out}(二进制，b9~b0)	温度(二进制)
50	500	200	00 11001000	00110010
60	600	240	00 11110000	00111100
70	700	300	01 00101100	01001011
80	800	320	01 01000000	01010000
90	900	360	01 01101000	01011010
100	1000	400	01 10010000	01100100

图 13-14 画出了 PIC18F458 和温度传感器的连接。注意，这里使用 LM336-2.5 齐纳二极管在 10 kΩ 电阻上得到稳定的 2.5 V 电压。使用 LM336－2.5 可以克服电源的波动。

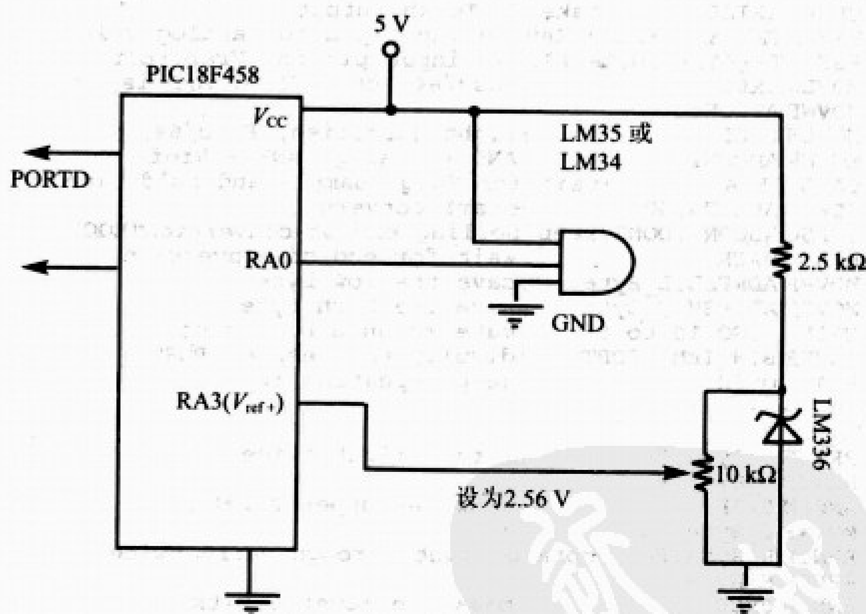


图 13-14　PIC18F458 和温度传感器的连接

例 13-10　对于表 13-9，验证 70℃的 PIC 输出值。计算 PIC18 A/D 寄存器 ADRESL 和 ADRESH 的值。

解：

因为 V_{ref}＝2.56 V，得到步长为 2.56 V/1024＝2.5 mV。

当温度是 70℃时，根据 LM34 的每一度输入对应着 10 mV 的变化量，可以得到 700 mV 的输出。于是步数＝700 mV/2.5 mV＝300（十进制）。300＝0100101100（二进制），因此，PIC18 A/D 输出寄存器 ADRESL 的值为 00101100，ADRESH 的值为 00000001。

522～523

13.4.4　温度的读取和显示

程序 13-4 和程序 13-4C 分别是温度读取和显示的汇编语言和 C 语言代码。

这些程序对应于图13-14。在这两个程序中，要注意以下几点。

(1) 将LM34(或LM35)连接到通道0(RA0引脚)。

(2) 将通道AN3(RA3引脚)连接到V_{ref}，大小为2.56 V。这样ADCON1寄存器的PCFG=0010。

(3) A/D的10位输出要除以4才能得到真实的温度值。

所使用的算法是这样的：将ADRESL右移2位；将ADRESH循环移位2位；对ADRESH和ADRESL进行逻辑或运算，得到8位的温度值输出。

程序 13-4

```
;Program 13-4
;this program reads the sensor and displays it on PORTD
L_Byte      SET 0x20   ;set a location 0x20 for L_Byte
H_Byte      SET 0x21   ;set a location 0x21 for H_Byte
BIN_TEMP    SET 0x22   ;set a location 0x22 for BIN_TEMP
       CLRF  TRISD         ;make PORTD an output
       BSF   TRISA,0 ;make RA0 an input pin for analog volt
       BSF   TRISA,3 ;make RA3 an input pin for Vref volt
       MOVLW 0x81               ;Fosc/64, channel 0, A/D is on
       MOVWF ADCON0
       MOVLW 0xC5               ;right justified, Fosc/64,
       MOVWF ADCON1             ;AN0 = analog, AN3 = Vref+
OVER   CALL  DELAY         ;wait for Tacq (sample and hold time)
       BSF   ADCON0,GO          ;start conversion
BACK   BTFSC ADCON0,DONE;keep polling end-of-conversion(EOC)
       BRA   BACK               ;wait for end-of-conversion
       MOVFF ADRESL,L_Byte      ;save the low byte
       MOVFF ADRESH,H_Byte      ;save the high byte
       CALL  ALGO_10_to_8       ;make it an 8-bit value
       MOVFF BIN_TEMP,PORTD     ;display the temp on PORTD
       BRA   OVER               ;keep repeating it
;-----------------
ALGO_10_to_8
       RRNCF L_Byte,F           ;rotate right twice
       RRNCF L_Byte,W
       ANDLW 0x3F               ;mask the upper 2 bits
       MOVWF L_Byte
       RRNCF H_Byte,F      ;rotate right through carry twice
       RRNCF H_Byte,W
       ANDLW 0xC0               ;mask the lower 6 bits
       IORWF L_Byte,W           ;combine low and high
       MOVWF BIN_TEMP
       RETURN
;------------
```

程序 13-4C

```
//Program 13-4C
void main(void)
{
   unsigned char L_Byte,H_Byte,Bin_Temp;
   TRISD=0;              //make PORTD output port
   TRISAbits.TRISA0=1; //RA0 = INPUT for analog input
  TRISAbits.TRISA2=1; //RA2 = INPUT for vref input
  ADCON0 = 0x81;        //Fosc/64, channel 0, A/D is on
  ADCON1 = 0xC5;        //right justified, Fosc/64,
                        //AN0 = analog, AN3 = Vref+
  while(1)
  {
    MSDelay(1);         //give A/D channel time to sample
    ADCON0bits.GO = 1;      //start converting
```

```
        while(ADCON0bits.DONE == 1);      //wait for EOC
        L_Byte=ADRESL;    //save the low byte
        H_Byte=ADRESH;    //save the high byte
        L_Byte>>=2;       //shift right
        L_Byte&=0x3F;     //mask the upper 2 bits
        H_Byte<<=6;       //shift left 6 times
        H_Byte&=0xC0;     //mask the lower 6 bits
        Bin_Temp= L_Byte|H_Byte;
        PORTD=Bin_Temp;
    }
}
```

13.4.5 复习题

1. 判断对错:传感器在连接到 ADC 之前要经过信号调整电路。
2. 对于每一________(华氏,摄氏)度,LM35 对应的提供________ mV 的电压。
3. 对于每一________(华氏,摄氏)度,LM34 对应的提供________ mV 的电压。
4. 当模拟输入引脚连接到 LM35 时,为什么要把 PIC 的 V_{ref} 设为 2.56 V?
5. 对于第 4 题的条件,如果 ADC 输出为 0011 1001,那么对应的温度是多少?

小结

本章介绍了 PIC 同现实设备(如 DAC 芯片、ADC 芯片和传感器等)的接口。首先,讨论了并行和串行 ADC 芯片,然后描述了 PIC18 内部的 ADC 模块是如何工作的,以及如何使用汇编语言和 C 语言对它编程。接着,介绍了 DAC 芯片以及它与 PIC 的接口。在最后一节中,学习了传感器,还讨论了模拟世界和数字设备之间的关系,以及数据捕获系统中常用的信号调整技术。

习题

1. 判断对错:传感器的输出是模拟量。
2. 判断对错:10 位的 ADC 有 10 位数字输出。
3. 判断对错:ADC0848 是 8 位 ADC。
4. 判断对错:MAX1112 是 10 位 ADC。
5. 判断对错:一个 8 通道模拟输入的 ADC 必须有 8 个引脚,每个引脚对应一个模拟输入。
6. 判断对错:串行 ADC 需要更长的时间来把转换好的数据送出芯片。
7. 判断对错:ADC0848 有 4 个模拟输入通道。
8. 判断对错:MAX1112 有 8 个模拟输入通道。
9. 判断对错:ADC0848 是串行 ADC。
10. 判断对错:MAX1112 是并行 ADC。
11. 下面哪个 ADC 有最佳的分辨率?
    (a) 8 位 (b) 10 位 (c) 12 位 (d) 16 位 (e) 以上的都具有
12. 对于第 11 题,哪一个 ADC 的步长最小?
13. 假设 V_{ref}=5 V,计算下面 ADC 的步长:

(a)8位 (b)10位 (c)12位 (d)16位

14. 假设V_{ref}=1.28 V,计算下面输出的对应V_{in}:

(a)D7~D0=11111111 (b)D7~D0=10011001 (c)D7~D0=1101100

15. 对于ADC0848,如果要得到5 mV的步长,那么V_{ref}的值应该设为多少?

16. 假设V_{ref+}=2.56,$V_{ref}-$=Gnd,计算下面输出的对应V_{in}:

(a)D7~D0=11111111 (b)D7~D0=10011001 (c)D7~D0=01101100

17. 判断对错:PIC18F452/458有片内的A/D转换器。

18. 判断对错:PIC18的A/D是8位ADC。

19. 判断对错:PIC18F452/458有8个模拟输入通道。

20. 判断对错:在PIC18F452/458中未使用的模拟引脚可以用作I/O引脚。

21. 判断对错:PIC18F452/458的A/D转换速度由晶振频率决定。

22. 判断对错:在上电复位时,PIC18F452/458的A/D模块已经打开,并且就绪。

23. 判断对错:PIC18F452/458的A/D模块有一个外部引脚用作转换开始信号。

24. 判断对错:PIC18F452/458的A/D模块每次只可以转换一个通道上的数据。

25. 判断对错:PIC18F452/458的A/D模块任何时候都可以有不同的V_{ref}+值。

26. 判断对错:PIC18F452/458的A/D模块可以使用V_{DD}作为V_{ref}+。

27. PIC18的A/D是怎样处理转换后的数据的?如何知道ADC已经准备好输出数据?

28. 在重新开始转换数据、而又未读取前一个数据时,PIC18的A/D怎么处理旧的数据呢?

29. 假设V_{ref}=Gnd。对于PIC18的A/D,计算对应于下面V_{ref}+的步长:

526

(a)V_{ref}=1.024 V (b)V_{ref}=2.048 V (c) V_{ref}=2.56 V

30. 对于PIC18,如果要得到2 mV的步长,V_{ref}的值应该为多少?

31. 对于PIC18,如果要得到3 mV的步长,V_{ref}的值应该为多少?

32. 假设步长为1 mV,当所有输出位都是1时,对应输入模拟电压是多少?

33. 假设V_{ref}=1.024 V,计算下面输出对应的V_{in}:

(a)D9~D0=0011111111 (b)D9~D0=0010011000 (c)D9~D0=0011010000

34. 对于PIC18的A/D,如果要得到4 mV的步长,那么V_{ref}的值应该为多少?

35. 假设V_{ref}+=2.56 V,V_{ref}−=Gnd,计算下面输出对应的V_{in}:

(a)D9~D0=1111111111 (b)D9~D0=1000000001 (c)D9~D0=1100110000

36. 假设XTAL=8 MHz,计算下面各选项对应的转换时间:

(a)F_{osc}/2 (b)F_{osc}/4 (c)F_{osc}/8 (d)F_{osc}/16 (e)F_{osc}/32

37. 假设XTAL=12 MHz,计算下面各选项对应的转换时间:

(a)F_{osc}/8 (b)F_{osc}/16 (c)F_{osc}/32 (d)F_{osc}/64

38. 如何启动PIC18的数据转换?

39. 如何知道PIC18的数据转换已经完成?

40. PIC18F452/458最少有________个模拟输入通道。

41. PIC18F452/458中有多少个端口用作模拟通道?

42. PIC18的哪个寄存器用作选择A/D通道的数目?

43. PIC18的哪个寄存器用作选择A/D转换速度?

44. PIC18的哪个寄存器用作选择模拟输入通道?

45. 如果要求F_{osc}/8、通道0、ADON打开,那么ADCON0寄存器的值是多少?

46. 如果要求F_{osc}/64、通道3模拟输入、右对齐输出,那么ADCON1寄存器的值是多少?

47. 如果要求 $F_{osc}/2$、通道 2、ADON 关闭，那么 ADCON0 寄存器的值是多少？
48. 如果要求 $F_{osc}/32$、通道 2 模拟输入、使用外部电源作 $V_{ref}+$、左对齐输出，那么 ADCON1 寄存器的值是多少？
49. 请说出 PIC18F452/458 的 A/D 中断标志位，并指出它们属于哪个寄存器。
50. 在上电复位时，PIC18F452/458 的 A/D 被分配的是________(低，高)优先级。
51. 判断对错：DAC0808 和 DAC1408 相同。
52. 请确定下面 n 位 DAC 可提供的离散电平数：
    (a)$n=8$ (b)$n=10$ (c)$n=12$
53. 如果 DAC1408 的 $I_{ref}=2$ mA，当所有输入位都为高电平时，请指出在什么情况下 I_{out} 等于 1.99 A。
54. 假设 DAC0808 的 $I_{ref}=2$ mA，请计算下面输入对应的 I_{out}：
    (a)10011001 (b)11001100 (c)11101110
    (d)00100010 (e)00001001 (f)10001000

527

55. 为了得到更小的步长，DAC 需要________(更多，更少)的数字输入。
56. 为了得到最大的输出，DAC 的输入应该为多少？
57. 传感器的线性输出是什么意思？
58. LM34 传感器对于每一度温度，提供________ mV 的电压。
59. 什么是信号调整？
60. 在图 13-14 中，在 V_{ref} 引脚连接的 LM336 齐纳二极管的作用是什么？

复习题答案

13.1 节

1. 电平数和 V_{ref} 电压 2. 8 3. 正确。 4. (a)8 (b)8 5. 1.28 V/256=5 mV
6. (a) 0.7/5mV=140(十进制)，D7～—D0=10001100(二进制)。
   (b) 1/5 mV=200(十进制)，D7～D0=11001000(二进制)。

13.2 节

1. V_{ref} 2. 10 3. 错误。 4. 错误。 5. 1 mV
6. (a)700 mV(1010111100) (b)1000 mV(1111101000)
7. (a)2 通道 (b)6 通道 8. 正确。 9. 1.6。
10. ADCON0 寄存器的 DONE 位

13.3 节

1. 数字，模拟 2. 模拟，数字 3. 8 4. (a)电流 (b)正确

13.4 节

1. 正确。 2. 摄氏，10 3. 华氏，10
4. 使用 10 位 ADC 的 8 位，可以提供 256 个电平数，于是 2.56 V/256=10 mV。LM35 每一摄氏度对应于 10 mV，和 ADC 的步长相吻合。
5. 00111001=57，所以是 57 度。

528

第 14 章 用闪存与 EEPROM 存储数据

学习目标：

- ☐ 各种半导体存储器的容量、组织结构和访问时间
- ☐ 芯片存储地址空间、数据引脚数和芯片容量之间的关系
- ☐ 闪存 ROM 及其在 PIC18 系统上的应用
- ☐ PROM、EPROM、UV-EPROM、EEPROM、闪存 EPROM 和掩模 ROM
- ☐ 向 PIC18 闪存写入数据的 PIC18 汇编和 C 语言编程
- ☐ 擦除闪存数据的 PIC18 汇编和 C 语言编程
- ☐ PIC18 EEPROM 存储器的数据写入
- ☐ PIC18 EEPROM 存储器的数据读取

这一章将讨论如何访问 PIC18F 的闪存和 EEPROM 存储器里的数据。在 14.1 节中，将学习半导体存储器的相关概念，而重点是不同种类的只读存储器(ROM)；14.2 节将讨论如何将数据写入 PIC18F 的闪存；14.3 节将探讨 PIC18 的 EEPROM 访问。

14.1 半导体存储器

本节将讨论各种半导体存储器及其特性，如容量、组织结构和访问时间。在设计微控制器系统时，主要是使用半导体存储器来存放程序和数据。半导体存储器直接与 CPU 相连，是 CPU 首先获取信息(程序和数据) 的存储器。因此，半导体存储器也称为主存储器。常用的半导体存储器有 ROM 和 RAM。在讨论 RAM 和 ROM 之前，先介绍几个重要的专业术语，如容量(capacity)、组织(organization) 和速度(speed)。

14.1.1 存储容量

半导体存储器芯片能存储的位数被称为芯片容量，单位可以是 Kbit(kilobit)、Mbit(megabit)等。这必须要同电脑系统的存储容量区分开来。IC 芯片的存储容量通常以位(bit) 给出，而电脑系统的存储容量通常以字节(byte) 给出。例如，技术杂志上的文章一篇报道说，128M 芯片已成为主流。在这种情况下，即使没提到其含义，也应该明白 128M 指的是 128 megabit。因为这篇文章指的是 IC 存储芯片。然而，如果一则广告报道说一台电脑配置有 128M 内存，那么 128M 指的是 128 兆字节，因为这里指的是电脑系统。

14.1.2 存储区组织

存储芯片内部是大量的存储地址单元。根据内部设计的不同，每个单元可以容纳 1 位、4

位、8位，甚至是16位。存储单元能够容纳的位数通常等于芯片的数据引脚数。那么存储芯片内部到底有多少个存储单元呢?这取决于地址引脚的数量。地址单元数等于2的n次方，其中n为地址引脚的数目。因此，存储芯片能够提供的存储位数等于位置数乘以每个单元的数据位数，现总结如下。

(1) 存储芯片的地址单元数是2^x，其中x表示地址引脚数。

(2) 每个地址单元包含y位，其中y表示数据引脚数。

(3) 整个芯片包含$2^x \times y$位，其中x为芯片地址引脚数，y为数据引脚数。

530

14.1.3 速度

数据访问速度是存储芯片最重要的特征指标之一。当访问数据时，地址出现在地址引脚上，READ引脚有效，然后等待一定时间，数据出现在数据引脚上。等待的时间越短越好，然而，价格也就越高。存储器芯片的速度通常被称为访问时间。根据芯片设计和制造过程所采用IC工艺的不同，存储器芯片的访问时间也从几纳秒到几百纳秒不等。

存储器的这3个重要特点(容量、组织和访问时间)将在本章进行详尽的讨论。表14-1可用作计算存储器特性指标的一个参考。例14-1和例14-2阐释了这些概念。

表14-1 2的幂指数

x	2^x	x	2^x
10	1K	19	512K
11	2K	20	1M
12	4K	21	2M
13	8K	22	4M
14	16K	23	8M
15	32K	24	16M
16	64K	25	32M
17	128K	26	64M
18	256K	27	128M

例14-1 一片给定的存储器芯片分别有12个地址引脚和4个数据引脚，请确定：

(a) 存储区组织 (b) 存储器容量

解：

(a) 这片存储芯片有4096个单元($2^{12} = 4096$)，每个单元可容纳4位数据。该芯片的存储区组织为4096×4，通常表示为4K×4。

(b) 存储器容量等于16 Kbit。因为总共有4K的单元，每个单元可容纳4位的数据。

例14-2 一片512K的存储芯片有8个数据引脚，请确定：

(a) 存储区组织 (b) 芯片的地址引脚数

解：

(a) 该存储芯片有8个数据引脚，因此芯片内部的每个单元可容纳8位数据。要确定芯片内存储单元的数目，只要将存储容量除以数据引脚数就可以了，即512K/8 = 64K。因此，该芯片的存储区组织为64K×8。

531

(b) 该芯片的地址线是 16 位。因为 $2^{16}=64K$。

14.1.4 ROM

ROM(只读存储器)是一种在断电后也不丢失内容的存储器。因此,ROM 也被称作非易失性存储器。只读存储器有很多种,如 PROM、EPROM、EEPROM、flash ROM 和掩模 ROM。下面分别介绍它们。

14.1.5 PROM 和 OTP

PROM(可编程 ROM)指的是一类允许用户烧录信息的 ROM。也就是说,PROM 是用户可编程存储器。在 PROM 中,每一位都有一条熔丝,在编程时将相应的熔丝烧断即可。如果烧进 PROM 的信息有错误,那么只有丢弃这片 PROM,因为里面的熔丝已被永久性地烧断。因此,PROM 也通常称为 OTP(一次性编程)。可编程 ROM(也即烧录 ROM)需要特殊的设备来编程,如 ROM 烧录器或 ROM 编程器。

14.1.6 EPROM 与 UV-EPROM

PROM(可擦除可编程 ROM)一旦被写入程序,就不能再改变,而 EPROM 可以解决这个问题。程序员可以在 EPROM 芯片上编程和擦除几千次。这在开发微控制器项目原型时是非常必要的。常用的 EPROM 是 UV-EPROM,其中 UV 代表紫外线。UV-EPROM 的问题是擦除过程需要 20 分钟。UV-EPROM 芯片有一个窗口,程序员可以透过这个窗口照射紫外线,擦除芯片上的数据。因此,EPROM 通常指紫外线可擦除 EPROM,或者简称为 UV-EPROM。图 14-1 给出了一片 UV-EPROM 的芯片引脚。

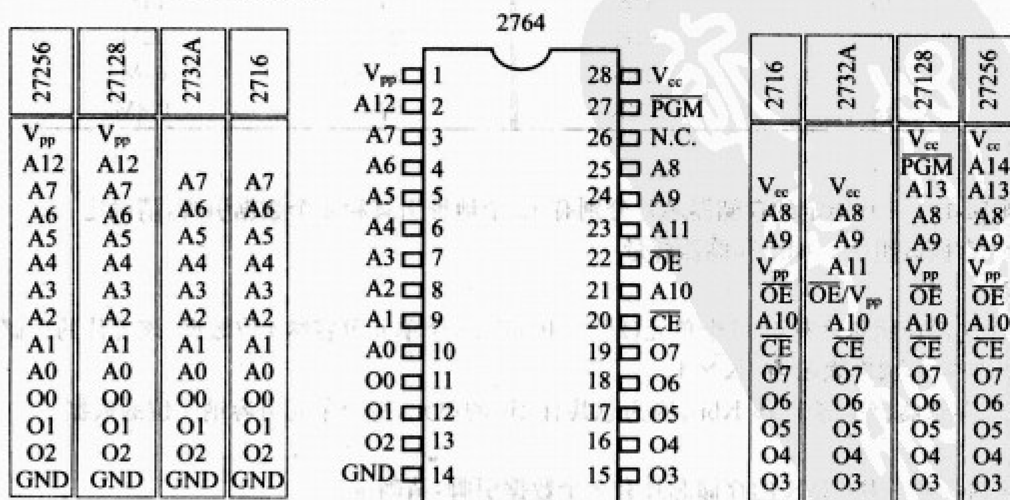


图 14-1 27xx ROM 系列芯片的引脚配置

对 UV-EPROM 芯片编程,必须执行下面的步骤。

(1) 先擦除芯片的内容。将芯片从系统板的插座上取出,放到 EPROM 擦除设备上,将其暴露在紫外线下 15 ~ 20 分钟。

(2) 编程。将芯片放入ROM烧录器(编程器),ROM烧录器根据芯片类型的不同,用12.5 V或更高的电压将代码或数据烧入芯片。烧录电压 V_{pp} 可在 UV-EPROM 的说明文件中查到。

(3) 将芯片放回系统板的插座上。

从上面的步骤可以看出,不仅需要 ROM 烧录器(编程器),而且需要单独的 EPROM 擦除设备。UV-EPROM 最主要的问题,同时也是致命的缺点是,它不能在系统板上擦除和编程。为解决这个问题,人们发明了 EEPROM。

注意表 14-2 给出的 IC 编号的格式。例如,芯片编号为 27128-25 表示该 UV-EPROM 芯片存储容量是 128 Kbit,访问时间为 250 ns。芯片编号表征了存储芯片的容量,访问时间按去掉一个 0 的形式给出。另外,C 代表 CMOS 工艺。注意 27xx 代表 UV-EPROM 系列芯片。

532

表 14-2 部分 UV-EPROM 芯片

型号	容量	组织	访问时间	引脚数	V_{pp}	
2716	16K	2K×8	450 ns	24	25 V	
2732	32K	4K×8	450 ns	24	25 V	
2732A-20	32K	4K×8	200 ns	24	21 V	
27C32-1	32K	4K×8	450 ns	24	12.5 V	CMOS
2764-20	64K	8K×8	200 ns	28	21 V	
2764A-20	64K	8K×8	200 ns	28	12.5 V	
27C64-12	64K	8K×8	120 ns	28	12.5 V	CMOS
27128-25	128K	16K×8	250 ns	28	21 V	
27C128-12	128K	16K×8	120 ns	28	12.5 V	CMOS
27256-25	256K	32K×8	250 ns	28	12.5 V	
27C256-15	256K	32K×8	150 ns	28	12.5 V	CMOS
27512-25	512K	64K×8	250 ns	28	12.5 V	
27C512-15	512K	64K×8	150 ns	28	12.5 V	CMOS
27C010-15	1024K	128K×8	150 ns	32	12.5 V	CMOS
27C020-15	2048K	256K×8	150 ns	32	12.5 V	CMOS
27C040-15	4096K	512K×8	150 ns	32	12.5 V	CMOS

例 14-3 请找出 ROM 芯片 27128 的数据和地址引脚数。

解:

27128 的存储容量为 128 Kbit。存储区组织为 16K×8(所有 ROM 都有 8 个数据引脚),表明有 8 个数据引脚和 14 个地址引脚(2^{14} = 16K)。

533

14.1.7 EEPROM

相比 EPROM 而言,EEPROM(electrically erasable programmable ROM,可电擦除可编程 ROM)有几个优点。比如擦除方式是电可擦除的,相对于 UV-EPROM 20 分钟的擦除时间,电擦除只需要很短的时间。另外,电擦除可以选择擦除的字节,而不像 UV-EPROM 那样,整片芯片的内容都被擦掉。EEPROM 的主要优点是,程序员可以在系统板上编程和擦除,而不需要将存储器芯片从插座上移走。也就是说,EEPROM 不像 UV-EPROM 那样需要外部的编程和擦除设备。为更好地利用 EEPROM,设计者必须将编程电路整合到系统板上。一般而言,EEPROM 每

一位的成本比UV-EPROM高得多。14.3节将介绍如何访问PIC18片上的EEPROM。

表14-3 部分EEPROM和FLASH芯片

型号	容量	组织	访问时间	引脚数	V_{PP}	
EEPROM						
2816A-25	16K	2K×8	250 ns	24	5 V	
2864A	64K	8K×8	250 ns	28	5 V	
28C64A-25	64K	8K×8	250 ns	28	5 V	CMOS
28C256-15	256K	32K×8	150 ns	28	5 V	
28C256-25	256K	32K×8	250 ns	28	5 V	CMOS
flash						
28F256-20	256K	32K×8	200 ns	32	12 V	CMOS
28F010-15	1024K	128K×8	150 ns	32	12 V	CMOS
28F020-15	2048K	256K×8	150 ns	32	12 V	CMOS

14.1.8 闪存EPROM

从20世纪90年代早期开始，flash EPROM作为用户可编程存储芯片，越来越受欢迎。首先，擦除整块芯片内容的时间不到1 s，正如其名字所示，闪存(flash memory)。另外，擦除方式是电擦除的，因此有时也称它为flash EEPROM。为避免混淆，通常称之为闪存。EEPROM与闪存的主要区别在于，在擦除(或写入)闪存的内容时，整块芯片都被擦除了，而对于EEPROM，程序员可以擦除任意的段或字节。近十年来，闪存的内容又被分成一个一个的数据块，擦除或写入时可以按块操作。与EEPROM不同的是，闪存不能按字节擦除或写入。因为闪存可以在系统板上编程，所以它已经取代UV-EPROM，用作PC机的BIOS ROM存储器。今天，闪存大量地应用于存储设备，如PDA、手机、U盘、MP3播放器等。一些计算机科学家相信，闪存将会取代硬盘成为大容量存储介质。这将会极大地提升电脑的性能，因为闪存是半导体存储器，访问时间是几百纳秒，而硬盘的访问时间是几十微秒。要实现这个目标，闪存的编程/擦除周期必须是无限次的，就像硬盘一样。编程/擦除周期指的是芯片在变得不可靠之前的可编程和可擦除的最大次数。现在，闪存和EEPROM的编程/擦除周期为100 000次，UV-EPROM是1000次，而RAM和硬盘是无限次的。

534

14.1.9 掩模ROM

掩模ROM指的是由IC制造商编程芯片内容的一种ROM。换句话说，它不是用户可编程ROM。掩模(mask)是IC制造工艺中的专业术语。由于烧录过程非常昂贵，所以只在需要大量地生产(几十万片)且可以确保芯片内容不再改变时，才用到掩模ROM。通常的做法是在项目的开发阶段使用UV-EPROM或闪存进行测试，当代码/数据完全确定之后才订做掩模版本的产品。掩模ROM的最大优点是比其他任何ROM都要便宜；可是一旦在代码/数据中发现错误，整批产品就都报废了。许多8051微控制器的生产厂家都支持掩模ROM版的8051。记住，所有ROM都是8个数据引脚，也就是说，存储区组织为x8。

14.1.10 RAM(随机访问存储器)

RAM也称为易失性存储器,因为断电会导致数据丢失。有时RAM指的是RAWM(读写存储器),这是与只读存储器ROM相对而言的。RAM分为三类:静态RAM(SRAM)、NV-RAM(非易失性RAM)和动态RAM(DRAM)。下面分别介绍它们。

14.1.11 SRAM

SRAM的存储单元由触发器组成,因此在保持数据时不需要刷新电路。这与下面将要讨论的DRAM恰好相反。使用触发器构成存储单元的问题是,每个单元需要至少6个晶体管构成,而且每个单元只能保存1位数据。最近几年,存储单元可以由4个晶体管构成了,但还是太多。4个晶体管的存储单元与COMS工艺配合使用,使得SRAM的容量有了大幅度的提升,但还是无法与DRAM相比。表14-4列举了几种SRAM。图14-2是一款SRAM芯片的引脚图。注意,在图14-2中,WE是写允许,OE是输出允许,分别控制写信号和读信号。

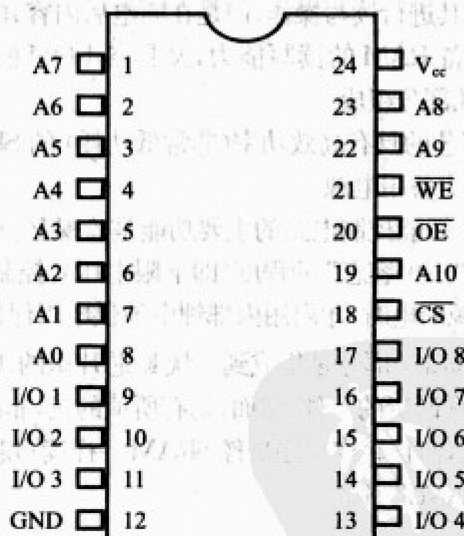


图 14-2 2K×8 SRAM 引脚

表 14-4 部分 SRAM 和 NV-RAM 芯片

SRAM					
型 号	容 量	组 织	访问时间	引脚数	V_{pp}
6116P-1	16K	2K×8	100 ns	24	CMOS
6116P-2	16K	2K×8	120 ns	24	CMOS
6116P-3	16K	2K×8	150 ns	24	CMOS
6116LP-1	16K	2K×8	100 ns	24	低功耗 CMOS
6116LP-2	16K	2K×8	120 ns	24	低功耗 CMOS
6116LP-3	16K	2K×8	150 ns	24	低功耗 CMOS
6264P-10	64K	8K×8	100 ns	28	CMOS
6264LP-70	64K	8K×8	70 ns	28	低功耗 CMOS

（续）

SRAM					
型　号	容　量	组　织	访问时间	引脚数	V_{PP}
6264LP-12	64K	8K×8	120 ns	28	低功耗 CMOS
62256LP-10	256K	32K×8	100 ns	28	低功耗 CMOS
62256LP-12	256K	32K×8	120 ns	28	低功耗 CMOS
Dallas 半导体公司的 NV-RAM					
型　号	容　量	组　织	访问时间	引脚数	V_{PP}
DS1220Y-150	16K	2K×8	150 ns	24	
DS1225AB-150	64K	8K×8	150 ns	28	
DS1230Y-85	256K	32K×8	85 ns	28	

14.1.12 NV-RAM

鉴于SRAM是易失性的，所以出现了一种非易失性的RAM，即NV-RAM。同其他RAM一样，NV-RAM也允许CPU对其进行读写操作，但是在断电后内容并不会丢失。NV-RAM整合了RAM和ROM的优点，既具备RAM的读写能力，又具备ROM的非易失性。为保存其内容，NV-RAM芯片内部由以下几部分组成。

(1) 使用采用CMOS工艺的具有高效功率(非常低功耗)的SRAM单元。

(2) 使用内部锂电池作为备用电源。

(3) 使用智能控制电路。智能控制电路的主要功能是监视V_{cc}引脚，检测外部电源的断电。如果V_{cc}引脚的电压下降到“不可容忍”的程度(即下限电压)，控制电路将自动切换到内部锂电池供电。当且仅当外部电源断电时，才启用内部锂电池供电以保持NV-RAM的内容。

536

这里必须强调的是，上面3个部分是集成到一块IC芯片里的。因此，就每一位的成本而言，非易失RAM是RAM中非常昂贵的一种。然而，物有所值的是，非易失性RAM可以在断电后保存芯片内容长达十年之久，而且允许程序员像SRAM一样读写芯片。表14-4列出了Dallas半导体公司生产的部分UV-RAM芯片。

14.1.13 DRAM

在计算机的早期时代，电脑设计者就非常希望有容量又大、价格又不贵的读写存储器。在1970年，Intel公司推出了第一款动态RAM。它的容量是1024位，一个电容存储一位。使用电容存储数据就减少了晶体管的数目。然而，由于电容漏电，所以需要连续刷新。这一点与SRAM相反。因为SRAM的每个位使用触发器，每个触发器需要6个晶体管，所以SRAM的存储单元体积大，于是密度较小。DRAM采用电容作为存储单元使得网状存储单元的体积更小。

DRAM的优缺点可总结如下。主要优点是容量大，每位成本小，耗电量低。缺点是必须周期性地刷新，因为电容的电荷会漏电。此外，在刷新DRAM时，是不允许访问其数据的。这与SRAM的触发器相反。SRAM的触发器只要有电源供电，就可以一直保持数据，而且不需要动态地刷新，其数据也是可以随时访问的。自1970年以来，DRAM的容量出现了爆炸性的增长。在1K(1024位)的芯片推出以后，1973年便出现了4K芯片，在1976年出现了16K芯片。20世纪80年代推出了64K、256K芯片，接着是1M和4M的存储芯片。在20世纪90年代则推出的是

16M、64M、256M，并开始出现1G的DRAM芯片。到了2000年，2Gbit成为工业标准。随着制造工艺越来越精细，大容量存储芯片将会源源不断地生产出来。记住，当谈论IC存储芯片的容量时，都假定以bit为单位。因此，1M芯片指的是1 megabit芯片，256K芯片指的是256 kilobit存储芯片。然而，当谈及电脑系统的存储器时，通常以B为单位。

14.1.14 DRAM的封装问题

DRAM存在一个问题，即如何将大量存储单元封装到一片只有几个引脚的芯片内。例如，一块64K(64K×1)的芯片必须有16条地址线和1条数据线，如果按照常规的方法，这就需要16个引脚来发送地址信号。这还不包含V_{cc}、接地和读/写控制引脚。采用常规方法进行数据访问，需要大量的引脚，这与IC设计者所钟爱的高密度小封装的设计理念是相悖的。为减少芯片地址的引脚数，可以采用多路复用/信号分离技术。这种方法将地址分成两个部分，用相同引脚分别发送两部分的地址，如表14-5所示。DRAM内部结构被分成用行和列表示的阵列。地址的前半部分称为行，后半部分称为列。例如，就存储区组织为64K×1的DRAM来说，在激活RAS(行地址选通)后，DRAM内部锁存器选中地址信号的前半部分，因此，地址前半部分通过A0～A7这8个引脚发送出去。接下来，在激活CAS(列地址选通)后，DRAM内部锁存器选中地址信号的后半部分，于是通过相同的引脚发送地址的后半部分。这样，地址信号8个引脚，加上RAS和CAS，共10个引脚，就替代了没有多路复用时所需的16个引脚。在访问DRAM里的数据时，必须提供行地址和列地址。为实现这一方法，在DRAM电路的外部需要一个2选1的多路选择器，在芯片内部需要有一个多路分离器。由于DRAM的接口(RAS和CAS，用作多路复用器以及用于刷新电路)非常复杂，所以市面上有专门的DRAM控制器。然而，很多小型微控制器项目不需要太多的RAM(通常少于64 KB)，所以使用EEPROM或NV-RAM的SRAM就足够了。 537

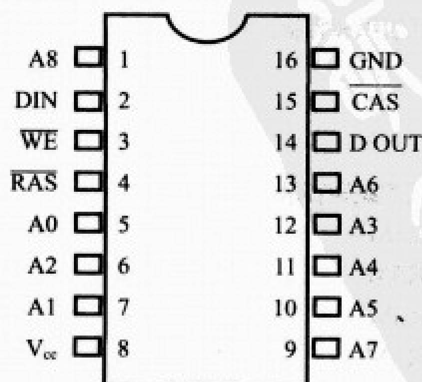


图14-3 256K×1的DRAM

14.1.15 DRAM存储区组织

在讨论ROM时，注意到所有的ROM都是8个数据引脚。然而DRAM存储器芯片则不同，它有x1、x4、x8、x16等存储区组织。如例14-4所示。

表 14-5　常用的 DRAM

型　号	速　度	容　量	组　织	引脚数
4164-15	150 ns	64K	64K×1	16
41464-8	80 ns	256K	64K×4	18
41256-15	150 ns	256K	256K×1	16
41256-6	60 ns	256K	256K×1	16
414256-10	100 ns	1M	256K×1	20
511000P-8	80 ns	1M	1M×1	18
514100-7	70 ns	4M	4M×1	20

例 14-4　讨论下面的存储器芯片的地址引脚数：

(a) 16K×4 DRAM　(b) 16K×4 SRAM

解：

由于 $2^{14}=16K$：

(a) DRAM 有 7 个地址引脚(A0～A6) 和两个选通引脚(RAS 和 CAS)。

538

(b) SRAM 有 14 个地址引脚，没有 DRAM 专用的 RAS 和 CAS。两类芯片都有 4 个引脚用作数据总线。

存储芯片的数据引脚也叫作 I/O。一些 DRAM 芯片有单独的 D_{in} 和 D_{out} 引脚。图 14-3 是一片 256K×1 DRAM 芯片的引脚图，包括地址引脚 A0～A8、RAS 和 CAS、WE(写允许)、数据输入(D IN) 和数据输出(D OUT)、电源(V_{cc}) 和地线(GND)。

14.1.16　复习题

1. 半导体存储器的速度在________以内。

   (a)μs　(b)ms　(c)ns　(d)ps

2. 请确定下面各片 ROM 的存储区组织和容量。

   (a) 地址引脚 14，数据引脚 8

   (b) 地址引脚 16，数据引脚 8

   (c) 地址引脚 12，数据引脚 8

3. 请确定下面各片 RAM 的存储区组织和容量。

   (a) 地址引脚 11，数据引脚 1，SRAM

   (b) 地址引脚 13，数据引脚 4，SRAM

   (c) 地址引脚 17，数据引脚 8，SRAM

   (d) 地址引脚 8，数据引脚 4，DRAM

   (e) 地址引脚 9，数据引脚 1，DRAM

   (f) 地址引脚 9，数据引脚 4，DRAM

4. 请确定下面各片存储器芯片的地址和数据引脚数。

   (a) 16K×4 SRAM　(b) 32K×8 EPROM　(c) 1M×1 DRAM

   (d) 256K×4 SRAM　(e) 64K×8 EEPROM　(f) 1M×4 DRAM

5. 下面哪些是易失性存储芯片？

   (a) EEPROM　(b) SRAM　(c) DRAM　(d) NV-RAM

14.2 PIC18F 只读闪存的擦写

PIC18F 有 3 类存储器：(a)SRAM，(b)Flash，(c)EEPROM。SRAM 用作通用存储器，包括贯穿本书的功能寄存器。EEPROM 只用来存储数据。闪存主要用来存储程序(代码)，有时也用来存储固定数据(如查询表)。在第 6 章已经讨论了怎样使用 TBLRD 指令读取程序闪存里的固定数据，在这一节将讨论怎样向闪存写入数据。而在下一节将讨论怎样访问 PIC18 的 EEPROM。

有两种方法可以向闪存写入程序或数据，或擦除其内容：(a) 使用外部闪存编程器(烧录器)(如 PICSTART)，(b) 使用指令(如 TBLWR)。这一节将介绍怎样使用指令 TBLWR(写表指令) 写闪存，同时还会介绍擦除闪存内容的方法。TBLRD 与 TBLWR 非常相似，因此有必要先阅读 6.3 节，在那里介绍了怎样使用 TBLRD 读取只读闪存里的数据。 539

14.2.1 使用 TBLWR 向闪存写入数据

TBLRD 与 TBLWR 有许多相同之处。回忆第 6 章使用 TBLRD 时，用寄存器 TBLPTR 作为指向闪存数据的指针，用 TABLAT 暂存取出来的数据。同样地，TBLWR 将 TABLAT 里的数据写入 TBLPTR 指向的只读闪存单元。在自增 / 自减的计算方面，TBLRD 与 TBLWR 完全一样。如表 14-6 所示。

表 14-6 PIC18 的写表指令

指 令	功 能	描 述
TBLWT *	写表	写后，TBLPTR 不变
TBLWT * +	后增量写表	写后，TBLPTR 加 1
TBLWT * −	后减量写表	写后，TBLPTR 减 1
TBLWT + *	前增量写表	TBLPTR 加 1 后，再写

指令 TBLRD 与 TBLWR 有着很大的区别。TBLRD 是读闪存中的单个字节，而 TBLWR 是写 8B 的数据块到闪存内。TBLRD 每次读取一个字节的数据到 TABLAT，也就是说，在读人下一个数据之前，必须先保存 TABLAT 的数据。TBLWR 使用了所谓的短写和长写向闪存写入数据。短写指的是使用 TBLWR 向 8 个 TABLAT 寄存器写人8 B数据块，每次一个字节。在短写完成之后，使用长写将整个数据块存人闪存。长写是在寄存器 EECON1 的帮助下完成的，如图 14-4 所示。注意，EECON1 也可用于 EEPROM，这将在下一节介绍。对照闪存和 EEPTROM 之间的差别。闪存的擦写过程按数据块为单位进行，而 EEPROM 可以每次擦写一个字节，也就是说，EEPROM 是字节可访问的存储器。闪存与 EEPROM 的读操作都是按字节进行的。闪存的数据块大小因闪存大小和用途而异。例如，PIC184580 的擦写数据块是 8 B，而其他闪存可能是 64 B 或 256 B。将 PIC18F 闪存分解成 8 B 的数据块，意味着存储器地址必须是以 8 B 为边界的。也就是说，如果只读闪存单元的地址是 A21 ～ A0，那么其低三位必须是 0。如图 14-5 所示。 540

EEPGD	CFGS	--	FREE	WRERR	WERN	WR#	RD#

EEPGD 程序闪存或数据 EEPROM 存储器的选择位
　　1 = 访问程序闪存
　　0 = 访问数据 EEPROM 存储器

CFGS　程序闪存 / 数据 EE 或者配置选择位
　　1 = 访问配置寄存器
　　0 = 访问程序闪存或数据 EEPROM 存储器

FREE　闪存的行擦除使能位
　　1 = 在下一个写命令时，擦除 TBLPTR 指向的程序存储器行(擦除操作完成后，该位清零)
　　0 = 仅进行写操作

WERR 写操作错误标志位
　　1 = 写操作被提早终止
　　0 = 写操作完成

WREN 写使能位
　　1 = 允许写周期
　　0 = 禁止向 EEPROM 或闪存写入

WR#　写控制位。低电平有效，用于闪存和 EEPROM。只能通过软件将其置为高电平。当写周期完成后，PIC 自动地将其置为低电平。
　　1 = 启动闪存或 EEPROM 的写循环(也可用于启动擦除过程)
　　0 = 写循环完成

RD#　读控制位。低电平有效，仅用于 EEPROM。只能通过软件将其置为高电平。当读周期完成后，PIC 自动将其置为低电平。
　　1 = 启动 EEPROM 的读循环
　　0 = 不启动读过程

541

图 14-4　EECON1(EEPROM 控制寄存器，也可用于闪存)

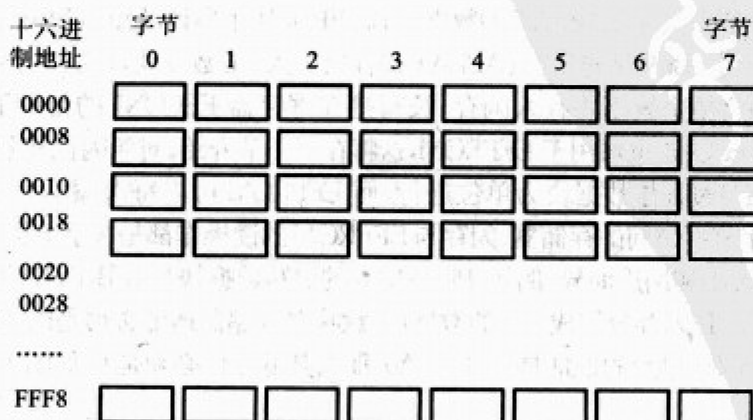


图 14-5　闪存的 8- 字节边界

虽然有8个TABLAT寄存器可用于短写，但这8个寄存器是不可以单独访问的。它们处于芯片的内部，当使用指令TBLWRT时，它们仅用于短写操作。比较图14-6和图14-7，对比闪存的读写过程。

14.2.2 写闪存的步骤

假定闪存的内容已被擦除，可以按下面的步骤将8 B数据块写入闪存。

(1) 将要写入首字节的地址赋给TBLPTR。

(2) 使用指令TBLWR，将8 B数据依次写入TABLAT。此时，短写结束。

(3) 为写操作设置EECON1寄存器，即设置EEPGD = 1，CFGS = 0，WREN = 1

(4) 使用BCF INTCON，GIE，禁止全局中断。

(5) 将55H写入EECON2哑元寄存器。此时，长写开始。

(6) 将AAH写入EECON2哑元寄存器。

(7) 使用BSF EECON1，WE将WR# 置1。在WE = 1时，写周期开始。

(8) 将8 B数据写入闪存需要2 ms。在写的工程中，CPU暂停工作，不允许读取操作码。在写过程完成后，WE# 自动地回到低电平。此时，长写结束。

(9) 使用BSF INTCON，GIE使能全局中断。

注意，从第(4)步开始必须禁止所有的中断，避免写过程(长写)出现中断。如果向闪存写入数据时被复位引脚(MCLR)或WDT(watch dog timer，看门狗计时器)中断，EECON1的WERR位将变为高电平，这表明写操作提早结束。幸运的是，EECON1的EEPGD位仍是高电平，允许通过重写数据来修正错误。EECON2在物理上是不存在的，也不能被访问，它仅在擦/写闪存/EEPROM存储器时起作用。程序14-1介绍了怎样将8 B数据写入起始地址为400H的闪存单元；写完之后，将数据读出并在PORTB上显示，每次显示一个字节，以此来验证写操作。

程序14-1的C语言版将在本节的最后给出。

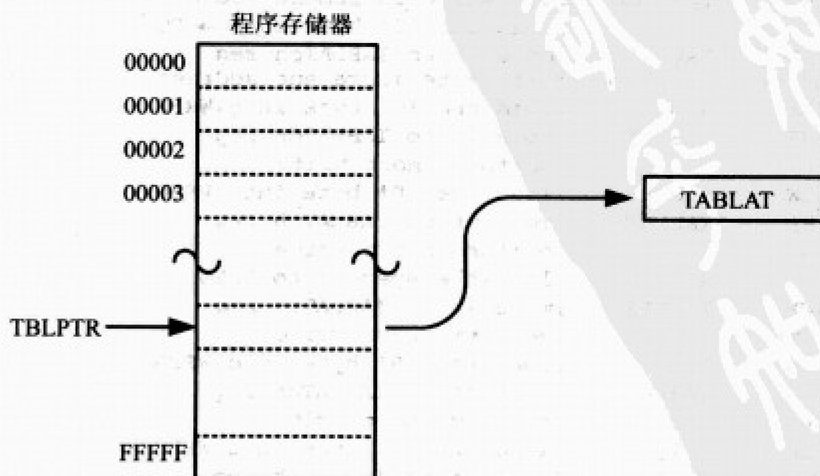


图14-6 读闪存

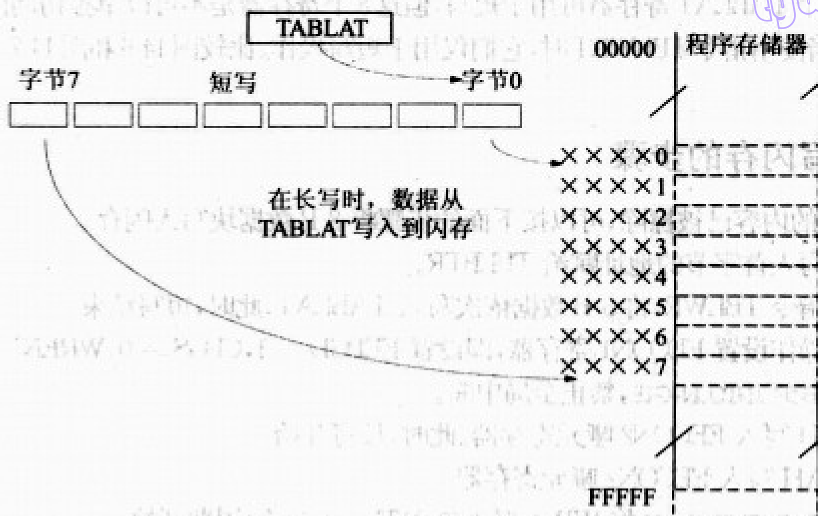


图 14-7 写闪存

543 程序14-1将GOOD BYE写入从400H开始的闪存存储单元中，读出数据并将其放在PORTB，每次一个字节。

程序 14-1

```
;Program 14-1
COUNT      EQU  0x20
      MOVLW     0x00
      MOVWF     TBLPTRL    ;load the low byte of address
      MOVLW     0x04
      MOVWF     TBLPTRH;load the high byte of address
      ;start a short write
      MOVLW     A'G'       ;load the 'G' byte into WREG
      MOVWF     TABLAT     ;move it to TABLATch reg
      TBLWT*+   ;perform short write increment address
      MOVLW     A'O'       ;load the 'O' byte into WREG
      MOVWF     TABLAT     ;move it to TABLATch reg
      TBLWT*+   ;perform short write increment address
      MOVLW     A'O'       ;load the 'O' byte into WREG
      MOVWF     TABLAT     ;move it to TABLATch reg
      TBLWT*+              ;perform short write
      MOVLW     A'D'       ;load the 'D' byte into WREG
      MOVWF     TABLAT     ;move it to TABLATch reg
      TBLWT*+              ;perform short write
      MOVLW     A' '       ;load the space into WREG
      MOVWF     TABLAT     ;move it to TABLATch reg
      TBLWT*+              ;perform short write
      MOVLW     A'B'       ;load the 'B' byte into WREG
      MOVWF     TABLAT     ;move it to TABLATch reg
      TBLWT*+              ;perform short write
      MOVLW     A'Y'       ;load the 'Y' byte into WREG
      MOVWF     TABLAT     ;move it to TABLATch reg
      TBLWT*+              ;perform short write
      MOVLW     A'E'       ;load the 'E' byte into WREG
      MOVWF     TABLAT     ;move it to TABLATch reg
      TBLWT*+              ;perform short write
```

```
;start the long write cycle (write to Flash itself)
        MOVLW       0x00
        MOVWF       TBLPTRL    ;load the low byte of address
        MOVLW       0x04
        MOVWF       TBLPTRH;load the high byte of address
        BSF         EECON1,EEPGD    ;point to Flash memory
        BCF         EECON1,CFGS     ;access Flash program
        BSF         EECON1,WREN     ;enable write
        BCF         INTCON,GIE      ;disable all interrupts
        MOVLW       55H             ;wreg = 55h
        MOVWF       EECON2          ;write to dummy reg
        MOVLW       0AAH            ;wreg = aah
        MOVWF       EECON2          ;write to dummy reg
        BSF         EECON1,WR       ;now write it to Flash
        NOP                         ;wait
        BSF         INTCON,GIE      ;enable all interrupts
        BCF         EECON1,WREN     ;disable write to memory
;read them back one byte at a time and examine the
;bytes on PORTB
        MOVLW       0x00
        MOVWF       TBLPTRL;reload the low byte of address
        MOVLW       0x04
        MOVWF       TBLPTRH;reload the high byte of address
        CLRF        TRISB     ;PORTB an output port
        MOVLW       0x8       ;counter = 8
        MOVWF       COUNT
OVER    TBLRD*+     ;read the byte to TABLAT and increment
        MOVFF       TABLAT,PORTB    ;send it to PORTB
        CALL        DELAY     ;wait enough to see the byte
        DECF COUNT,F          ;decrement counter
        DECFSZ      COUNT,F
        BRA         OVER      ;continue for all the bytes
```

544

如果数据块不是 8 B,那么数据块未使用部分的值将不受影响。如程序 14-2 所示。

程序 14-2 将 HELLO 写入从 450H 开始的闪存存储单元中,读出数据并将其放在 PORTB,每次一个字节。

程序 14-2

```
;Program 14-2
COUNT        EQU   0x20
        MOVLW       0x50
        MOVWF       TBLPTRL    ;load the low byte of address
        MOVLW       0x04
        MOVWF       TBLPTRH;load the high byte of address
        ;start a short write
        MOVLW       A'H'       ;load the 'H' byte into WREG
        MOVWF       TABLAT     ;move it to TABLATch reg
        TBLWT*+     ;perform short write and increment
        MOVLW       A'E'       ;load the 'E' byte into WREG
        MOVWF       TABLAT     ;move it to TABLATch reg
        TBLWT*+     ;perform short write and increment
        MOVLW       A'L'       ;load the 'L' byte into WREG
```

```
      MOVWF     TABLAT    ;move it to TABLATch reg
      TBLWT*+    ;perform short write
      MOVLW     A'L'      ;load the 'L' byte into WREG
      MOVWF     TABLAT    ;move it to TABLATch reg
      TBLWT*+    ;perform short write
      MOVLW     A'O'      ;load the 'O' byte into WREG
      MOVWF     TABLAT    ;move it to TABLATch reg
      TBLWT*+    ;perform short write
;start the long write cycle (write to Flash itself)
      MOVLW     0x50
      MOVWF     TBLPTRL   ;load the low byte of address
      MOVLW     0x04
      MOVWF     TBLPTRH  ;load the high byte of address
      BSF       EECON1,EEPGD   ;point to Flash memory
      BCF       EECON1,CFGS    ;access Flash program
      BSF       EECON1,WREN    ;enable write
      BCF       INTCON,GIE     ;disable all interrupts
      MOVLW     55H            ;wreg = 55h
      MOVWF     EECON2         ;write to dummy reg
      MOVLW     0AAH           ;wreg = aah
      MOVWF     EECON2         ;write to dummy reg
      BSF       EECON1,WR      ;now write it to Flash
      NOP                      ;wait
      BSF       INTCON,GIE     ;enable all interrupts
      BCF       EECON1,WREN    ;disable write to memory
;read them back one byte at a time and examine the
;bytes on PORTB
      MOVLW     0x50
      MOVWF     TBLPTRL ;reload the low byte of address
      MOVLW     0x04
      MOVWF     TBLPTRH;reload the high byte of address
      CLRF      TRISB          ;PORTB an output port
      MOVLW     0x05           ;counter = 5
      MOVWF     COUNT
OVER  TBLRD*+             ;read the byte and increment
      MOVFF     TABLAT,PORTB   ;send it to PORTB
      CALL      DELAY     ;wait enough to see the byte
      DECF COUNT,F        ;dec counter
      DECFSZ    COUNT,F
      BRA       OVER      ;continue for all the bytes
```

程序14-3将PIC18程序存储区内的数据块传送到RAM,然后将RAM中的数据写入Flash,从新的闪存单元读取数据并传到PIC18的串行端口,每次一个字节。

程序 14-3

```
;Program 14-3
COUNT      EQU   0x0B
BUFRAM     EQU   0x20
      MOVLW D'8'          ;number of bytes to retrieve
      MOVWF COUNT
      MOVLW high (BUFRAM) ;point to buffer
      MOVWF FSR0H
```

```
     MOVLW low (BUFRAM)
     MOVWF FSR0L
     MOVLW upper (CODE_DATA)  ;load TBLPTR
     MOVWF TBLPTRU
     MOVLW high (CODE_DATA)
     MOVWF TBLPTRH
     MOVLW low (CODE_DATA)
     MOVWF TBLPTRL
;retrieve the data from program memory
READ_BLOCK
     TBLRD*+           ;read into TABLAT, and increment
     MOVF TABLAT, W            ;get data
     MOVWF POSTINC0            ;store data
     DECFSZ COUNT              ;done?
     BRA READ_BLOCK            ;repeat
     MOVLW upper (NEW_DATA)    ;load TBLPTR
     MOVWF TBLPTRU
     MOVLW high (NEW_DATA)
     MOVWF TBLPTRH
     MOVLW low (NEW_DATA)
     MOVWF TBLPTRL
     MOVLW high (BUFRAM)       ;point to buffer
     MOVWF FSR0H
     MOVLW low (BUFRAM)
     MOVWF FSR0L
     MOVLW 8                   ;number of bytes in RAM
     MOVWF COUNT
;move the data back to program memory
WRITE_BACK
     MOVF POSTINC0, W          ;get a byte from RAM
     MOVWF TABLAT   ;store the byte in table latch
     TBLWT*+                   ;perform a short write
     DECFSZ COUNT          ;loop until buffers are full
     BRA WRITE_BACK
     MOVLW upper (NEW_DATA) ;load TBLPTR
     MOVWF TBLPTRU
     MOVLW high (NEW_DATA)
     MOVWF TBLPTRH
     MOVLW low (NEW_DATA)
     MOVWF TBLPTRL
     BSF EECON1, EEPGD ;point to Flash program memory
     BCF EECON1, CFGS     ;access Flash program memory
     BSF EECON1, WREN     ;enable write to memory
     BCF INTCON, GIE      ;disable interrupts
     MOVLW 55h            ;write 55h
     MOVWF EECON2
     MOVLW 0AAh           ;write 0AAh
     MOVWF EECON2         ;start program (CPU stall)
     BSF EECON1, WR
     NOP
     BSF INTCON, GIE      ;re-enable interrupts
     BCF EECON1, WREN     ;disable write to memory
```

```
;read them back one byte at a time and send serially
      BSF   TRISD,7          ;PORTD.7 as in input
      MOVLW 0x20   ;enable transmit and low baud rate
      MOVWF      TXSTA       ;write to reg
      BCF        PIR1,TXIF
      MOVLW      D'15'       ;9600 bps (Fosc/(64*Speed)-1)
      MOVWF      SPBRG       ;write to reg
      BCF   TRISC, TX ;make TX pin of PORTC an output
      BSF   RCSTA, SPEN ;enable the entire serial port
      MOVLW 8                ;number of bytes in RAM
      MOVWF COUNT
      MOVLW upper (NEW_DATA) ;load TBLPTR
      MOVWF TBLPTRU
      MOVLW high (NEW_DATA)
      MOVWF TBLPTRH
      MOVLW low (NEW_DATA)
      MOVWF TBLPTRL
      CLRF TRISB             ;PORTB an output port
      MOVLW      0x8         ;counter = 8
      MOVWF      COUNT
LN    TBLRD*+                ;read the character
      MOVF       TABLAT,W
R1    BTFSS PIR1, TXIF ;wait until the last bit is gone
      BRA   R1               ;stay in loop
      MOVWF      TXREG ;load the value to be transmitted
      DECFSZ COUNT           ;loop until buffers are full
      BRA   LN               ;repeat
```

14.2.3 擦除闪存的步骤

尽管可以使用外部闪存编程器来擦除闪存的内容，但是 PIC18 允许编程擦除闪存内容。擦除过程是按数据块工作的，而不是按字节工作。用于擦除的数据块最小为64 B。这就是说，为使它们变为 64 B 数据块的边界，地址的低 6 位全为 0。将闪存内 64 B 数据块擦除的步骤如下。

(1) 把要擦除数据块的地址赋给 TBLPTR。

(2) 为擦除操作设置 EECON1 如下：

(a) EEPGD = 1， (b)CFGS = 0， (c)WREN = 1， (d)FREE = 1。

(3) 使用 BCF INTCON,GIE 屏蔽所有的中断。

548

(4) 将 55H 写入 EECON2 哑元寄存器。

(5) 将 AAH 写入 EECON2 哑元寄存器。

(6) 使用语句 BSF EECON1, WE 将 WR# 置 1。当 WE = 1 时，擦除过程开始。

(7) 擦除 64 B数据块需要 2 ms。在擦除过程中，CPU 暂停工作，不允许读取操作码。在擦除过程完成后，WE# 自动地回到高电平。

(8) 使用语句 BCF INTCON,GIE 重新使能中断。

程序 14-4 说明了怎样擦除 64 B 的数据块。

程序 14-4

```
;Program 14-4: This program erases the Flash
;memory starting at location 0x500.
      ORG  0
      MOVLW      upper(MYDATA)
      MOVWF      TBLPTRH          ;load the upper address
      MOVLW      high(MYDATA)
      MOVWF      TBLPTRH   ;load the high byte of address
      MOVLW      low(MYDATA)
      MOVWF      TBLPTRL   ;load the low byte of address
      BSF        EECON1,EEPGD     ;point to Flash memory
      BCF        EECON1,CFGS      ;access Flash program
      BSF        EECON1,WREN      ;enable write
      BSF   EECON1, FREE   ;enable row erase operation
      BCF        INTCON,GIE       ;disable all interrupts
      MOVLW      55H              ;wreg = 55h
      MOVWF      EECON2           ;write to dummy reg
      MOVLW      0AAH             ;wreg = aah
      MOVWF      EECON2           ;write to dummy reg
      BSF        EECON1,WR        ;now write it to Flash
      NOP                         ;wait.
      BSF        INTCON,GIE       ;enable all interrupts
      BCF        EECON1,WREN      ;disable write to memory
HERE BRA         HERE
      ORG 500H
MYDATA data "ABCDEFGH"
      END
```

考察程序 14-5,它包括闪存的擦除、写入和读取操作。

程序 14-5

```
;Program 14-5: This program erases the message of
;"GOOD BYE" from Flash addresses 0x1200 and replaces
;it with "HELLO".
      MOVLW      upper(MYDATA)
      MOVWF      TBLPTRU    ;load the upper address
      MOVLW      high(MYDATA)
      MOVWF      TBLPTRH   ;load the high byte of address
      MOVLW      low(MYDATA)
      MOVWF      TBLPTRL    ;load the low byte of address
      BSF        EECON1,EEPGD     ;point to Flash memory
      BCF        EECON1,CFGS      ;access Flash program
      BSF        EECON1,WREN      ;enable write
      BSF   EECON1, FREE   ;enable row erase operation
      BCF        INTCON,GIE       ;disable all interrupts
      MOVLW      55H              ;wreg = 55h
      MOVWF      EECON2           ;write to dummy reg
      MOVLW      0AAH             ;wreg = aah
      MOVWF      EECON2           ;write to dummy reg
      BSF        EECON1,WR        ;now write it to Flash
      NOP                         ;wait
      BSF        INTCON,GIE       ;enable all interrupts
      BCF        EECON1,WREN      ;disable write to memory
```

```
        MOVLW     upper(MYDATA)
        MOVWF     TBLPTRU           ;load the upper address
        MOVLW     high(MYDATA)
        MOVWF     TBLPTRH  ;load the high byte of address
        MOVLW     low(MYDATA)
        MOVWF     TBLPTRL   ;load the low byte of address
;start a short write
        MOVLW     A'H'              ;load the byte into WREG
        MOVWF     TABLAT            ;move it to TABLATch reg
        TBLWT*+   ;perform short write and increment
        MOVLW     A'E'              ;load the byte into WREG
        MOVWF     TABLAT            ;move it to TABLATch reg
        TBLWT*+   ;perform short write and increment
        MOVLW     A'L'              ;load the byte into WREG
        MOVWF     TABLAT            ;move it to TABLATch reg
        TBLWT*+                     ;perform short write
        MOVLW     A'L'              ;load the byte into WREG
        MOVWF     TABLAT            ;move it to TABLATch reg
        TBLWT*+                     ;perform short write
        MOVLW     A'O'              ;load the byte into WREG
        MOVWF     TABLAT            ;move it to TABLATch reg
        TBLWT*+                     ;perform short write
;start the long write cycle (write to Flash itself)
        BSF       EECON1,EEPGD      ;point to Flash memory
        BCF       EECON1,CFGS       ;
        BSF       EECON1,WREN       ;enable write
        BCF       INTCON,GIE        ;disable all interrupts
        MOVLW     55H               ;wreg = 55h
        MOVWF     EECON2            ;write to dummy reg
        MOVLW     0AAH              ;wreg = aah
        MOVWF     EECON2            ;write to dummy reg
        BSF       EECON1,WR         ;now write it to Flash
        NOP                         ;wait
        BSF       INTCON,GIE        ;enable all interrupts
        BCF       EECON1,WREN       ;disable write to memory
;read them back one byte at a time and examine the
;bytes on PORTB
        MOVLW     upper(MYDATA)
        MOVWF     TBLPTRU           ;load the upper address
        MOVLW     high(MYDATA)
        MOVWF     TBLPTRH  ;load the high byte of address
        MOVLW     low(MYDATA)
        MOVWF     TBLPTRL   ;load the low byte of address
        CLRF TRISB                  ;PORTB an output port
        MOVLW     0x05              ;counter = 5
        MOVWF     COUNT
OVER    TBLRD*+   ;read byte to TABLAT and point to next
        MOVFF     TABLAT,PORTB   ;send it to PORTB
        CALL      DELAY ;wait enough to see byte on PORTB
        DECF      COUNT,F           ;decrement counter
        BNZ       OVER     ;continue for all the bytes
```

```
      ORG 1200H
MYDATA data "GOOD BYE"
      END
```

14.2.4 闪存擦写操作的C语言编程

程序14-6C到程序14-8C是前几个程序的C语言版。

程序 14-6C

```
/*Program 14-6C: This C program (a) writes the mes-
sage "GOOD BYE" to Flash memory starting at location
400H, (b) reads the data from Flash and places it in
PORTB one byte at a time. */

#include <p18Cxxx.h>
void Delay(unsigned int itime);

void main()
{
     unsigned char x;
     //write to program memory
     TBLPTR = (short long)0x0400;   //load TBLPTR
     TABLAT='G';                    //load in TABLAT
     _asm TBLWTPOSTINC _endasm      //short write
     TABLAT='O';                    //load in TABLAT
     _asm TBLWTPOSTINC _endasm      //short write
     TABLAT='O';                    //load in TABLAT
     _asm TBLWTPOSTINC _endasm      //short write
     TABLAT='D';                    //load in TABLAT
     _asm TBLWTPOSTINC _endasm      //short write
     TABLAT=' ';                    //load in TABLAT
     _asm TBLWTPOSTINC _endasm      //short write
     TABLAT='B';                    //load in TABLAT
     _asm TBLWTPOSTINC _endasm      //short write
     TABLAT='Y';                    //load in TABLAT
     _asm TBLWTPOSTINC _endasm      //short write
     TABLAT='E';                    //load in TABLAT
     _asm TBLWTPOSTINC _endasm      //short write

     //long write
     TBLPTR = (short long)0x0400;   //reload TBLPTR
     EECON1bits.EEPGD=1;
     EECON1bits.CFGS=0;
     EECON1bits.WREN=1;
     INTCONbits.GIE=0;
     EECON2=0x55;
     EECON2=0xAA;
     EECON1bits.WR=1;
     _asm NOP _endasm
     INTCONbits.GIE=1;
     EECON1bits.WREN=0;
```

```
    //read from program memory send to PORTB
    TBLPTR = (short long)0x0400;  //reload TBLPTR
    for(x=0;x<8;x++){
          _asm TBLRDPOSTINC _endasm
          PORTB=TABLAT;
          Delay(250);
    }
}
```

程序　14-7C

```
//Program 14-7C: This C program erases the Flash
//memory starting at location 0x500.

#include <p18Cxxx.h>

#pragma romdata const_table = 0x500
const rom char my_const_array[10] = "GOOD BYE";
#pragma romdata
void main()
{

  //erase program memory
  TBLPTR = (short long)0x0500;     //load TBLPTR
  EECON1bits.EEPGD=1;
  EECON1bits.CFGS=0;
  EECON1bits.WREN=1;
  EECON1bits.FREE=1;
  INTCONbits.GIE=0;
  EECON2=0x55;
  EECON2=0xAA;
  EECON1bits.WR=1;
  _asm NOP _endasm
  INTCONbits.GIE=1;
  EECON1bits.WREN=0;
}
```

552

程序　14-8C

```
//Program 14-8C: This C program erases the message of
//"GOOD BYE" from Flash addresses 0x1200 and replaces
//it with "HELLO".

#include <p18Cxxx.h>

void Delay(unsigned int itime);

#pragma romdata const_table = 0x1200
const rom char my_const_array[10] = "GOOD BYE";
#pragma romdata
void main()
{
  unsigned char x;
```

```
//erase program memory
TBLPTR = (short long)0x1200;      //load TBLPTR
EECON1bits.EEPGD=1;
EECON1bits.CFGS=0;
EECON1bits.WREN=1;
EECON1bits.FREE=1;
INTCONbits.GIE=0;
EECON2=0x55;
EECON2=0xAA;
EECON1bits.WR=1;
_asm NOP _endasm
INTCONbits.GIE=1;
EECON1bits.WREN=0;

TBLPTR = (short long)0x1200;      //load TBLPTR
TABLAT='H';                       //load in TABLAT
_asm TBLWTPOSTINC _endasm         //short write
TABLAT='E';                       //load in TABLAT
_asm TBLWTPOSTINC _endasm         //short write
TABLAT='L';                       //load in TABLAT
_asm TBLWTPOSTINC _endasm         //short write
TABLAT='L';                       //load in TABLAT
_asm TBLWTPOSTINC _endasm         //short write
TABLAT='O';                       //load in TABLAT
_asm TBLWTPOSTINC _endasm         //short write

//long write
TBLPTR = (short long)0x1200;      //reload TBLPTR
EECON1bits.EEPGD=1;
EECON1bits.CFGS=0;
EECON1bits.WREN=1;
INTCONbits.GIE=0;
EECON2=0x55;
EECON2=0xAA;
EECON1bits.WR=1;
_asm NOP _endasm
INTCONbits.GIE=1;
EECON1bits.WREN=0;

//read from program memory send to PORTB
TBLPTR = (short long)0x1200;      //reload TBLPTR
for(x=0;x<8;x++){
  _asm TBLRDPOSTINC _endasm
  PORTB=TABLAT;
  Delay(250);
}
}
```

14.2.5 复习题

1. 判断对错：PIC18F 闪存可用于存储程序代码和数据。
2. 判断对错：PIC18F SRAM 可用于存储程序代码和数据。
3. 判断对错：在 PIC18F 中，不允许向闪存写入数据。
4. 判断对错：读闪存是按字节进行的，而写闪存是按数据块进行的。
5. 判断对错：在长写期间，CPU 保持取指令和执行指令的状态。
6. 在写入 PIC18F458 的闪存时，数据块的大小是________。

554

7. 在擦除 PIC18F458 的闪存时，数据块的大小是________。

14.3 PIC18 EEPROM 的数据读取和写入

PIC18 系列芯片大部分都有 EEPROM 存储器，容量从 256 B 到几千字节不等。例如，PIC18F4520 有 256 B的 EEPROM，而 PIC184585 有 1024 B。表 14-7 列出了几种型号的芯片和它们的 EEPROM 空间。PIC18 内的闪存可用于存储程序代码和数据，而 EEPROM 仅用于存储数据。对于 PIC18 的 3 个存储区，SRAM 和 EEPROM 仅用来存储数据，而闪存主要用来存储程序，有时也存储固定数据。如图 14-8 所示。

表 14-7 部分 PIC18 芯片的 EEPROM 容量

型　号	片上闪存	片上 RAM	片上 EEPROM
PIC18F1220	4 KB	256 B	256 B
PIC18F1230	4 KB	256 B	128 B
PIC18F2410	16 KB	768 B	0 B
PIC18F4520	32 KB	1536 B	256 B
PIC18F4580	32 KB	1536 B	256 B
PIC18F4585	48 KB	3328 B	1024 B

555

注意：片上 RAM 不包含 SFR 空间在内。

14.3.1 向 EEPROM 写入数据

与 EEPROM 有关的寄存器有 4 个，分别是：

(1) EEADR，8 位寄存器，用作指向 EEPROM 单元的指针；
(2) EEDATA，8 位寄存器，用来存放写入 EEPROM 的数据；
(3) EECON1，如图 14-4 所示，EEPROM 和闪存均要用到；
(4) EECON2，哑元寄存器，EEPROM 和闪存均要用到。

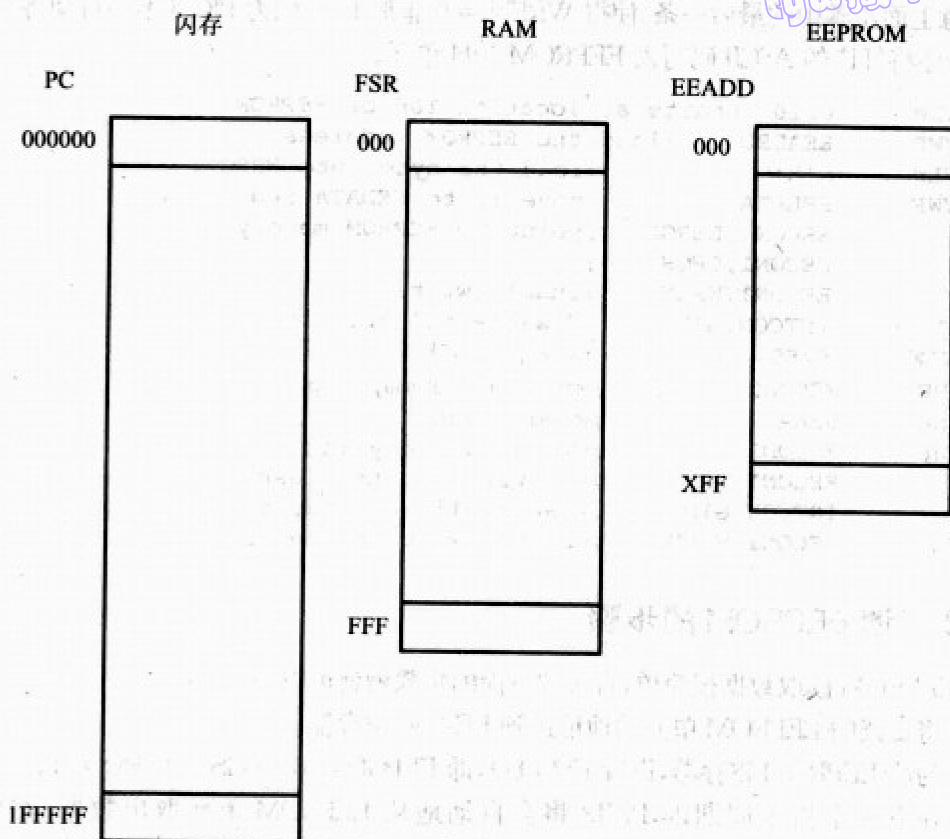


图 14-8 对比 PIC18F 的存储器

注意，EEADR(EE 地址)寄存器在 PIC18F452/458 中只有 8 位。8 位地址提供 256B 的空间，足以覆盖这些芯片内的 EEPROM。一些微控制器芯片如 PIC18F4585，有 1024B 的 EEPROM，因此，需要用 EEADR 的低字节(EEADRL)和高字节(EEADRH)地址来覆盖 EEPROM 存储区。

14.3.2 写 EEPROM 的步骤

将字节数据写入 EEPROM 存储单元，需要遵循以下步骤。

(1) 将要写入数据字节的 EEPROM 存储单元地址送入 EEADR 寄存器。

(2) 将数据字节送到 EEDATA 寄存器。

(3) 为写 EEPROM，需设置 EECON1，即 EEPGD＝0，CFGS＝0，WREN＝1。

(4) 使用 BCF INTCON，GIE 使能全局中断。

(5) 将 55H 写入 EECON2 哑元寄存器。

(6) 将 AAH 写入 EECON2 哑元寄存器。

(7) 使用 BSF EECON1，WE 将 WR# 置 1。在 WE＝1 时，写周期开始。

(8) 在写周期完成后，WE# 自动清零。

(9) 使用 BSF INTCON，GIE 重新使能全局中断。

(10) 将 WREN 清零，避免一些失控的程序突然改写 EEPROM。

注意上面步骤中的最后一条。使得 WREN = 0 非常重要，因为 PIC18 不会自动将其清零。下面的程序将‘H’的 ASCII 码写入 EEPROM 10H 单元。

```
MOVLW    0x10 ;starts at location 10H of EEPROM
MOVWF    EEADRD     ;load the EEPROM  address
MOVLW    A'H'          ;load the byte into WREG
MOVWF    EEDATA        ;move it to EEDATA reg
BCF      EECON1,EEPGD  ;point to EEPROM memory
BCF      EECON1,CFGS   ;
BSF      EECON1,WREN   ;enable write
BCF      INTCON,GIE    ;disable all interrupts
MOVLW    0x55          ;wreg = 55h
MOVWF    EECON2        ;write to dummy reg
MOVLW    0xAA          ;wreg = aah
MOVWF    EECON2        ;write to dummy reg
BSF      EECON1,WR     ;now write it to Flash
BSF      INTCON,GIE    ;enable all interrupts
BCF      EECON1,WREN   ;disable write to memory
```

556

14.3.3 读 EEPROM 的步骤

从 EEPROM 读取数据很简单，直接按下面的步骤做就可以。

(1) 将要读的 EEPROM 单元的地址送到 EEADR 寄存器。

(2) 为读 EEPROM 寄存器，设置 EECON1，即 EEPGD = 0，CFGS = 0，RD = 1。

(3) 在下一个指令周期内，PIC18 将会自动地从 EEPROM 单元取出数据，然后放入 EEDATA 寄存器。唯一要做的事是在下一个数据读入前，将 EEDATA 里的值转移到一个安全的地方。下面的代码是从 EEPROM 读入一个字节，然后放到 PORTB 中。

```
MOVLW    0x10        ;read location 10H of EEPROM
MOVWF    EEADR       ;load the EEPROM address
BCF      EECON1,EEPGD   ;point to EEPROM memory
BCF      EECON1,CFGS    ;
BSF      EECON1,RD      ;enable read
NOP  ;data is fetched from EEPROM to EEDATA reg
MOVFF    EEDATA,PORTB   ;place the data in PORTB
```

程序 14-9 将 HELLO 写入从地址 0 开始的 EEPROM 存储单元，从 EEPROM 读数据，然后将其放到 PORTB 中，每次一个字节。

程序 14-9

```
;Program 14-9: Writing to EEPROM
     MOVLW 0x0       ;starts at location 0H of EEPROM
     MOVWF EEADR     ;load the EEPROM address
     MOVLW A'H'      ;load the byte into WREG
     MOVWF EEDATA    ;move it to EEDATA reg
     CALL EE_WRT
     INCF EEADR,F    ;point to next location
     MOVLW A'E'      ;load the byte into WREG
     MOVWF EEDATA    ;move it to EEDATA reg
```

```
        CALL EE_WRT
        INCF EEADR,F    ;point to next location
        MOVLW A'L'      ;load the byte into WREG
        MOVWF EEDATA    ;move it to EEDATA reg
        CALL EE_WRT
        INCF EEADR,F    ;point to next location
        MOVLW A'L'      ;load the byte into WREG
        MOVWF EEDATA    ;move it to EEDATA reg
        CALL EE_WRT
        INCF EEADR,F    ;point to next location
        MOVLW A'O'      ;load the byte into WREG
        MOVWF EEDATA    ;move it to EEDATA reg
        CALL EE_WRT
        INCF EEADR,F    ;point to next location
;read EEPROM one byte at a time and send it to
;PORTB
        MOVLW 0x0 ;starts at location 0H of EEPROM
        MOVWF EEADR             ;load the EEPROM address
        BCF   EECON1,EEPGD      ;point to EEPROM memory
        BCF   EECON1,CFGS       ;
        MOVLW 0x05              ;count = 5
        MOVWF COUNT
        CLRF TRISB              ;make PORTB output port
OVER BSF  EECON1,RD             ;enable read
        NOP
        MOVFF EEDATA,PORTB      ;read the data to PORTB
        CALL DELAY              ;wait
        INCF EEADR,F            ;point to next location
        DECF COUNT,F            ;decrement counter
        BNZ  OVER               ;keep repeating
HERE BRA  HERE
EE_WRT
        BCF  EECON1,EEPGD       ;point to EEPROM memory
        BCF  EECON1,CFGS        ;
        BSF  EECON1,WREN        ;enable write
        BCF  INTCON,GIE         ;disable all interrupts
        MOVLW 0x55              ;wreg = 55h
        MOVWF EECON2            ;write to dummy reg
        MOVLW 0xAA              ;wreg = aah
        MOVWF EECON2            ;write to dummy reg
        BSF  EECON1,WR          ;now write it to Flash
        BSF  INTCON,GIE         ;enable all interrupts
EE_WAIT  BTFSS PIR2,EEIF
        BRA  EE_WAIT
        BCF  PIR2,EEIF
        RETURN
```

程序 14-10 将 PIC18 芯片内程序存储区内的一数据块移入到 EEPROM 中，然后从 EEPROM 读出数据并将其通过 PIC18 的串行端口发送出去，每次一个字节。

程序 14-10

```
#include p18f458.inc

COUNT     EQU  0x0B
BUFRAM    EQU  0x20

      MOVLW D'8'              ;number of bytes to retrieve
      MOVWF COUNT
      MOVLW 0H          ;starts at location 0H of EEPROM
      MOVWF EEADR                ;load the EEPROM address
      MOVLW upper (CODE_DATA)   ;load TBLPTR
      MOVWF TBLPTRU
      MOVLW high (CODE_DATA)
      MOVWF TBLPTRH
      MOVLW low (CODE_DATA)
      MOVWF TBLPTRL
;retrieve the data from program memory
READ_BLOCK
      TBLRD*+           ;read into TABLAT, and increment
      MOVF TABLAT, W        ;get data
      MOVWF EEDATA          ;load data
      CALL EE_WRT           ;save data
      INCF EEADR,F          ;point to next location
      DECFSZ COUNT          ;done?
      BRA READ_BLOCK        ;repeat
;read them back one byte at a time and send to serial
;port
      BSF  TRISD,7          ;PORTD.7 as in input
      MOVLW 0x20 ;enable transmit and low baud rate
      MOVWF TXSTA           ;write to reg
      BCF   PIR1,TXIF
      MOVLW D'15'  ;9600 bps (Fosc / (64 * Speed) - 1)
      MOVWF SPBRG           ;write to reg
      BCF  TRISC, TX ;make TX pin of PORTC an output
      BSF  RCSTA, SPEN ;enable the entire serial port
      MOVLW 8               ;number of bytes in RAM
      MOVWF COUNT
      MOVLW      0x0  ;start at location 0H of EEPROM
      MOVWF      EEADR      ;load the EEPROM address
      CLRF TRISB            ;make PORTB an output port
      MOVLW      0x8        ;counter = 8
      MOVWF      COUNT
LN    CALL EE_RD            ;read the character
      CALL SENDCOM     ;send character to serial port
      INCF EEADR,F
      DECFSZ COUNT     ;loop until buffers are full
      BRA  LN               ;repeat
HERE BRA HERE
;-------------
SENDCOM
S1    BTFSS PIR1, TXIF ;wait unil the last bit is gone
      BRA S1          ;stay in loop
```

```
        MOVWF TXREG     ;load the value to be transferred
        RETURN          ;return to caller
;--------------
EE_WRT
        BCF   EECON1,EEPGD   ;point to EEPROM memory
        BCF   EECON1,CFGS    ;
        BSF   EECON1,WREN    ;enable write
        BCF   INTCON,GIE     ;disable all interrupts
        MOVLW 0x55           ;wreg = 55h
        MOVWF EECON2         ;write to dummy reg
        MOVLW 0xAA           ;wreg = aah
        MOVWF EECON2         ;write to dummy reg
        BSF    EECON1,WR     ;now write it to Flash
        BSF   INTCON,GIE     ;enable all interrupts
EE_WAIT BTFSS PIR2,EEIF
        BRA   EE_WAIT
        BCF   PIR2,EEIF
        RETURN
;--------------
EE_RD
        BCF EECON1, EEPGD    ;point to DATA memory
        BCF EECON1, CFGS
        BSF EECON1, RD       ;EEPROM read
        MOVF EEDATA, W       ;W = EEDATA
        RETURN
;--------------
        ORG 0x0300
CODE_DATA
        DATA "MOVE ME"
        END
```

560

14.3.4 使用 C 语言访问 EEPROM

程序 14-11C 介绍了怎样用 C 语言读和写 EEPROM。这是前面的一些程序的 C 语言版。

程序 14-11C 将 YES 写入 EEPROM 存储器，然后从 EEPROM 读出同样的数据，并将其发送到 PORTB，每次一个字节。

程序 14-11C

```
//Program 14-11C
#include <p18F458.h>
void EE_WRT(void);
unsigned char EE_READ(void);
void Delay(unsigned int itime);
void main()
{
   unsigned char x;
   TRISB=0;              //make PORTB output
//write to EEPROM
   EEADR=0x0;            //EEPROM location
   EEDATA='Y';           //write this char to it
   EE_WRT();
   EEADR=0x1;
```

```
  EEDATA='E';
  EE_WRT();
  EEADR=0x2;
  EEDATA='S';
  EE_WRT();
  EECON1bits.WREN=0; //disable write
//read from EEPROM and place it on PORTB
  EECON1bits.RD=1;    //enable read
  EEADR =0x0;         //EEPROM location
  x=EE_READ();        //read data from EEPROM
  PORTB=x;            //place it on PORTB
  Delay(250);
  EEADR =0x1;         //EEPROM location
  x=EE_READ();
  PORTB=x;            //place it on PORTB
  Delay(250);
  EEADR =0x2;         //EEPROM location
  x=EE_READ();
  PORTB=x;            //place it on PORTB
  while(1);
}
void EE_WRT()
{
  EECON1bits.EEPGD=0;//point to EEPROM
  EECON1bits.CFGS=0;
  EECON1bits.WREN=1; //enable write
  INTCONbits.GIE=0;  //disable interrupts
  EECON2=0x55;
  EECON2=0xAA;
  EECON1bits.WR=1;
  INTCONbits.GIE=1;
  while(!PIR2bits.EEIF);
  PIR2bits.EEIF=0;
}
unsigned char EE_READ()
{
  EECON1bits.EEPGD=0;
  EECON1bits.CFGS=0;
  EECON1bits.RD=1;
  return(EEDATA);
}
```

561

程序 14-12C 将数据块从闪存传到 RAM，将该数据块写入 EEPROM 存储器，然后从 EEPROM 读出相同的数据并将其发送到串行端口，每次一个字节。

程序 14-12C

```
//Program 14-12C
#include <p18f458.h>
void EE_WRT(void);
unsigned char EE_READ(void);
void SerTx(unsigned char);

void main(){
  rom far char* RomPointer="MOVE ME";
```

```
   char RamString[7];
   unsigned char x,ch,k=sizeof(RomPointer);
   TXSTA=0x20;        //choose low baud rate,8-bit
   SPBRG=15;          //9600 baud rate, XTAL = 10 MHz
   TXSTAbits.TXEN=1;
   RCSTAbits.SPEN=1;

   //move the string to RAM
   for(x=0;x<7;x++){
     RamString[x]=RomPointer[x];
   }

   //move the string to EEPROM
   for(x=0;x<7;x++){
     EEADR=x;
     EEDATA=RamString[x];
     EE_WRT();
     }
   EECON1bits.WREN=0;//disable write

 //read from EEPROM and send serially
   for(x=0;x<7;x++){
      EEADR=x;
      ch=EE_READ();
      SerTx(ch);
   }

   while(1);//infinite loop
 }

 void EE_WRT()
 {
      EECON1bits.EEPGD=0;  //point to EEPROM
      EECON1bits.CFGS=0;
      EECON1bits.WREN=1;   //enable write
      INTCONbits.GIE=0;    //disable interrupts
      EECON2=0x55;
      EECON2=0xAA;
      EECON1bits.WR=1;
      INTCONbits.GIE=1;
      while(!PIR2bits.EEIF);
      PIR2bits.EEIF=0;
  }

 unsigned char EE_READ()
 {
      EECON1bits.EEPGD=0;  //point to EEPROM
      EECON1bits.CFGS=0;
      EECON1bits.RD=1;
      return(EEDATA);
 }
```

562

```
void SerTx(unsigned char c)
{
    while(PIR1bits.TXIF==0);  //wait until transmitted
    TXREG=c;          //place character in buffer
}
```

14.3.5 复习题

1. 判断对错:PIC18 的 EEPROM 存储器可用于存储程序和数据。
2. 判断对错:PIC18F4580 有 1024 B 的 EEPROM 存储空间。
3. 判断对错:在 PIC18 中,在断电后 EEPROM 里的内容将会丢失。
4. 判断对错:在 PIC18 中,EEPROM 是读写存储器。
5. 判断对错:每片 PIC18 均有 1KB 的 EEPROM。
6. 与闪存相比,EEPROM 的优势是什么?

563

小结

本章描述了基于 PIC18 的系统的存储器接口。首先介绍的是半导体存储器的基础知识,接下来在存储容量、组织和访问时间 3 个方面对各种存储器进行了比较。

ROM(只读存储器)是非易失性存储器,常用来存储程序。本章介绍了各种 ROM 的相对优势,包括 PROM、EPROM、UV-EPROM、EEPROM、闪存 EPROM 和掩模 ROM。

RAM(随机访问存储器)主要用来存储数据或程序。本章介绍了各种 RAM 的相对优势,这包括 SRAM、NV-RAM、校验和字节 RAM 和 DRAM。

本章还讨论了 PIC18 的闪存空间,并给出了访问闪存的汇编语言程序和 C 语言程序。最后,探讨了 PIC18 芯片的 EEPROM 存储器,并介绍了访问 EEPROM 的汇编语言程序和 C 语言程序。

习题

1. 容量为 4 Mbit 的存储芯片和 4 MB 的电脑存储器有什么不同?
2. 判断对错:地址引脚越多,则芯片内存储单元就越多。(假定数据引脚的数量是固定的。)
3. 判断对错:数据引脚越多,则每个存储单元能容纳的位数就越多。
4. 判断对错:数据引脚越多,则芯片存储容量就越大。
5. 判断对错:数据引脚和地址引脚越多,则芯片存储容量就越大。
6. 存储芯片的速度指的是它的________。
7. 判断对错:存储芯片的价格根据其存储容量和访问时间的不同而不同。
8. EEPROM 相对于 UV-EPROM 的主要优势在于________。
9. 判断对错:SRAM 的存储单元容量比 DRAM 大。
10. 在 EPROM、DRAM 和 SRAM 中,哪种存储器必须周期性地刷新?
11. 哪种存储器适合用作 PC 的缓存?
12. 在 SRAM、UV-EPROM、NV-RAM、DRAM 中,哪些是易失性存储器?

13. RAS 和 CAS 与下列哪种存储器有关?
(a) EPROM (b) SRAM (c) DRAM (d) 全部
14. 下列哪种存储器需要外部的多路复用器?
(a) EPROM (b) SRAM (c) DRAM (d) 全部
15. 请确定下面各芯片的存储区组织和容量。
(a) EEPROM A0 ~ A14,D0 ~ D7 (b) UV-EPROM A0 ~ A12,D0 ~ D7
(c) SRAM A0 ~ A11,D0 ~ D7 (d) SRAM A0 ~ A12,D0 ~ D7
(e) DRAM A0 ~ A10,D0 (f) SRAM A0 ~ A12,D0
(g) EEPROM A0 ~ A11,D0 ~ D7 (h) UV-EPROM A0 ~ A10,D0 ~ D7
(i) DRAM A0 ~ A8,D0 ~ D8 (j) DRAM A0 ~ A7,D0 ~ D7

564

16. 请确定下列芯片的存储容量、地址和数据引脚。
(a) 16K×8 ROM (b) 32K×8 ROM (c) 64K×8 SRAM (d) 256K×8 EEPROM
(e) 64K×8 ROM (f) 64K×4 DRAM (g) 1M×8 SROM (h) 4M×4 DRAM
(i) 64K×8 NV-ROM
17. 判断对错:PIC18F 内的闪存主要是用于程序存储器。
18. 判断对错:PIC18F 内的闪存可用来存储固定数据。
19. 判断对错:PIC18F 用于存储程序的最大存储空间是 2 MB。
20. 判断对错:从闪存读取数据,可以一个字节一个字节地进行。
21. 判断对错:将数据写入闪存,可以一个字节一个字节地进行。
22. 判断对错:将数据写入闪存,必须按 64 B 的数据块进行。
23. 判断对错:擦除闪存,可以一个字节一个字节地进行。
24. 判断对错:在擦写闪存时,EECON2 寄存器是可选的。
25. 哪些寄存器用于读取闪存内的固定数据?
26. 哪些寄存器用于写入数据到闪存?
27. 哪些寄存器用于擦除闪存内容?
28. EECON1 的 WREN 与 WR 有什么不同?
29. TBLRD 使用了哪些寄存器?
30. TBLWRT 使用了哪些寄存器?
31. PIC18 短写(short write) 与长写(long write) 之间有什么不同?
32. 在哪个写(短写,长写) 操作期间,CPU 读取操作码被挂起?
33. PIC184580 写闪存时的数据块大小是________。
34. PIC184580 擦除闪存时的数据块大小是________。
35. 指出下列地址中有 8 B 边界的地址:
(a) 510H (b) 512H (c) 514H (d) 516H (e) 518H (f) 51AH (g) 51CH (h) 51EH
36. 指出下列地址中有 64 字节边界的地址:
(a) 500H (b) 520H (c) 540H (d) 560H (e) 580H (f) 5A0H (g) 5C0H

565

37. 指出 2000 ~ 2020H 可用于写闪存的地址边界。
38. 指出 2000 ~ 2100H 可用于擦除闪存的地址边界。
39. 编制程序,擦除闪存的某一段数据,然后将 Hello World 写进去。
40. 在 39 题中,编制程序,检验写操作,即:读取闪存,然后将数据发送到串行端口,每次一个字节。
41. 判断对错:PIC18F 的 EEPROM 主要用于程序代码的存储。

42. 判断对错:PIC18F的EEPROM只用于数据存储。
43. 判断对错:PIC18F的每个型号都至少有256 B的EEPROM。
44. 判断对错:从EEPROM读取数据可以按字节进行。
45. 判断对错:写数据进EEPROM可以按字节进行。
46. 判断对错:写数据进EEPROM必须按64 B数据块进行。
47. 判断对错:擦除EEPROM数据可以按字节进行。
48. 判断对错:在读和写EEPROM时,EECON2寄存器是可选的。
49. 判断对错:EECON2寄存器可用于闪存和EEPROM的写操作。
50. 从EEPROM读数据时,要用到哪些寄存器?
51. 向EEPROM写数据时,要用到哪些寄存器?
52. 指出PIC18中闪存与EEPROM之间主要有什么不同?
53. 在PIC18中向EEPROM写数据时数据块的大小是________。
54. 读EEPROM时要用到EECON1的哪些位?
55. 为什么在写闪存/EEPROM过程中必须要屏蔽所有中断?
56. 为什么在读闪存/EEPROM过程中不屏蔽所有中断?
57. 编制程序,将Hello World写入EEPROM。
58. 在57题中,编制程序,检验写操作,即:读取EEPROM,然后将数据发送到串行端口,每次一个字节。

复习题答案

14.1 节

1. c

566

2. (a) 16K×8,128 Kbit　(b) 64K×8,512 Kbit　(c) 4K×8,32 Kbit
3. (a) 2K×8,2 Kbit　(b) 8K×4,32 Kbit　(c) 128K×8,1 Mbit
   (d) 64K×4,256 Kbit　(e) 256K×1256 Kbit　(f) 256K×4,1 Mbit
4. (a) 64 Kbit,14个地址位和4个数据位
   (b) 256 Kbit,15个地址位和8个数据位
   (c) 1 Mbit,10个地址位和1个数据位
   (d) 1 Mbit,18个地址位和4个数据位
   (e) 512 Kbit,16个地址位和8个数据位
   (f) 4 Mbit,10个地址位和4个数据位
5. b,c

14.2 节

1. 正确。　2. 错误。　3. 错误。　4. 正确。　5. 错误。　6. 8B　7. 64B

14.3 节

1. 错误。　2. 错误。　3. 错误。　4. 正确。　5. 错误。

567

6. 在EEPROM中,可以写入单个字节的数据,而在闪存中,则必须写入数据块。

第 15 章 CCP 和 ECCP 编程

学习目标：

- ☐ PIC18 的比较和捕捉特征
- ☐ CCP 和 ECCP 模块定时器的使用
- ☐ CCP 和 ECCP 模块比较特征的工作
- ☐ CCP 和 ECCP 模块捕捉特征的工作
- ☐ 比较和捕捉特征的汇编编程和 C 语言编程
- ☐ 在 CCP 和 ECCP 模块中 PWM(脉冲宽度调制) 的工作
- ☐ PWM 创建的汇编编程和 C 语言编程

本章将讨论 PIC18 的捕捉 / 比较 / 脉冲宽度调制(即 CCP)。15.1 节将介绍标准型 CCP 模块和增强型 CCP 模块之间的区别。15.2 节将介绍 PIC18 的比较特征。15.3 节将学习 PIC18 的捕捉特征。PIC18 的脉冲宽度调制(PWM) 将在 15.4 节中讨论。15.5 节将概述 ECCP。本章所有小节都将使用汇编编程和 C 语言编程来说明 PIC18 的重要特征。

15.1 标准型和增强型 CCP 模块

根据芯片型号的不同，PIC18 内部有 0～5 个数量不等的 CCP 模块。这些 CCP 模块通常被命名为 CCP1、CCP2、CCP3、CCPx 等。近几年来，为了更好地控制直流电机，CCP 的 PWM 特征得到了很大的发展，产生了增强型 CCP(ECCP)。因此，给定的 PIC 系列芯片有两个标准的 CCP 模块和一个或多个 ECCP 模块；所有模块都位于一个单独的芯片上。如表 15-1 所示。ECCP 模块将在第 17 章讨论。

表 15-1 PIC1 的 CCP 和 ECCP 模块

芯片型号	CCP 的数量	ECCP 的数量
PIC18F2220	2	0
PIC18F4220	1	1
PIC18F452/4520	1	1
PIC18F458/4580	1	1
PIC18F65J10	2	3

15.1.1 CCP 和计时器

为了编程 CCP 模块，必须理解 PIC18 的定时器。在学习本章之前，最好先回顾一下第 9 章的定

时器。使用的CCP特征不同，则定时器的用法也不同。用于CCP特征的定时器分配如表15-2所示。

表15-2　PIC18定时器的用途

CCP模式	定时器
捕捉	定时器1或者定时器3
比较	定时器1或者定时器3
PWM	定时器2

注意：T3CON寄存器用于选择定时器的比较和捕捉模式。

15.1.2　CCP寄存器

每个CCP模块都有3个相关的寄存器，如下所示。

CCPxCON是一个8位控制寄存器。使用该寄存器可以选择比较、捕捉和PWM模式中的一种。如图15-1所示。

570

CCPRxL和CCPRxH分别构成16位寄存器的低字节和高字节。该16位寄存器可用作16位比较寄存器或16位捕捉寄存器，或是PWM的8位占空比寄存器，但不能同时使用。如图15-1和图15-2所示。CCP1CON寄存器是用来选择操作模式的。

-	-	DC1B1	DC1B2	CCP1M3	CCP1M2	CCP1M1	CCP1M0

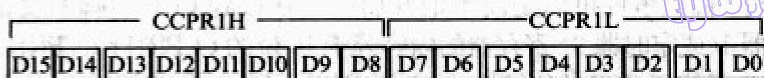
DC1B1 占空比第1位。仅用于PWM模式
10位占空比寄存器的第1位，用于PWM

DC1B0 占空比第0位。仅用于PWM模式
10位占空比寄存器的最低有效位(第0位)，用于PWM
CCPxL寄存器用作10位占空比寄存器的第2～9位

CCP1M3～CCP1M0　CCP1模式选择

0000	关闭CCP1
0001	保留
0010	比较模式。在匹配时，翻转CCP1的输出引脚(CCP1IF被置位)
0011	保留
0100	捕捉模式，在每个下降沿发生捕捉
0101	捕捉模式，在每个上升沿发生捕捉
0110	捕捉模式，在每4个上升沿发生捕捉
0111	捕捉模式，在每16个上升沿发生捕捉
1000	比较模式。初始化CCP1引脚为低电平，在比较匹配时，强制CCP1引脚为高电平。(CCP1IF被置位)
1001	比较模式。初始化CCP1引脚为高电平，在比较匹配时，强制CCP1引脚为低电平。(CCP1IF被置位)
1010	比较模式。在比较匹配时，产生软件中断(CCP1IF置位，CCP1引脚无影响)
1011	比较模式。特殊事件触发(CCP1IF置位，定时器1或定时器3复位为0)
11xx	PWM模式

图15-1　CCP1控制寄存器(该寄存器用来选择捕捉、比较和PWM三种操作模式之一)



571

图 15-2 CCP 的高低字节寄存器

15.1.3 CCP 引脚

每个CCP模块都分配有一个单独的引脚。因此，带有两个标准型CCP模块的PIC18系列芯片（如PIC18F65J10）有两个引脚，每个引脚对应着一个CCP模块。如图15-3所示。对于增强型CCP(ECCP)，虽然有一个单独的引脚，但是在使用ECCP的PWM特征时，将对该引脚的4个功能进行编程。这将在本章末和第17章讨论。

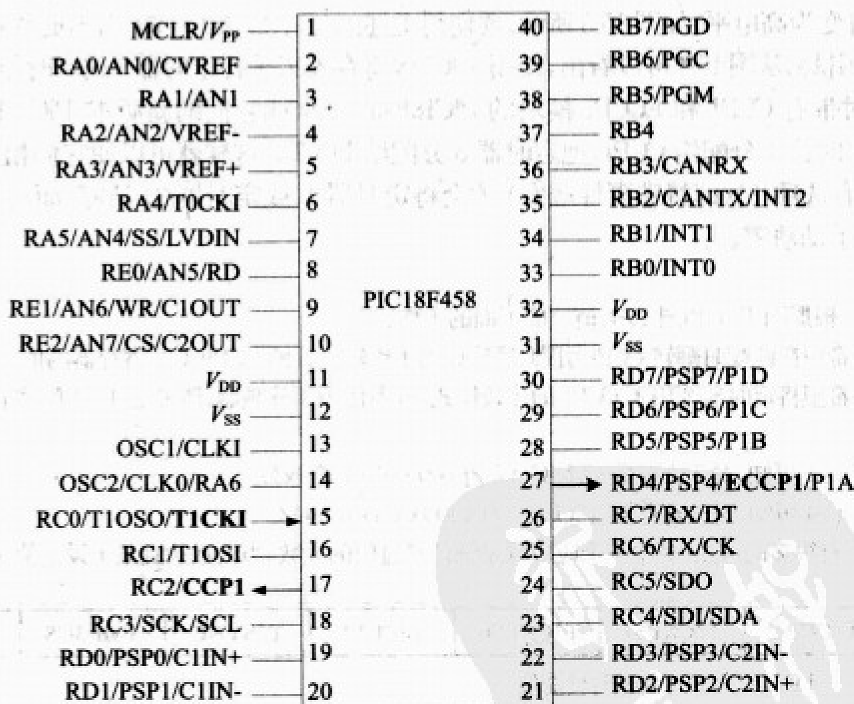


图 15-3 PIC18F458/4580(452/4520) 的标准 CCP 引脚

15.1.4 复习题

1. 判断对错：PIC18 芯片内部可集成多个 CCP 模块。
2. 判断对错：CCP1 是 16 位的寄存器。
3. 判断对错：每个标准 CCP 模块都有一个对应的引脚。
4. 指出 PIC18F452/458(或 PIC18F4520/4580) 芯片用于标准 CCP1 模块的引脚数。

572

15.2 比较模式编程

CCPxCON 寄存器的选择位可以用来选择 CCP 模块的比较模式。比较模式可以在微控制

器外部产生一个事件。该事件可以是简单地开启一个连接到CCP引脚的设备,或者是启动ADC转换。定时器1(或定时器3)寄存器的内容等于16位的CCPR1H:CCPR1L寄存器的内容时,事件被触发。为了使用CCP的比较模式,必须用相同的初始值赋值给16位寄存器(CCP1H:CCP1L)和定时器1(或定时器3)寄存器。当定时器1(或定时器3)连续计数时,它的值将与CCPR1H:CCPR1L的值进行比较。如果两者相等,CCP1引脚将会执行下列动作之一:

(1) 驱动CCP1引脚为高电平

(2) 驱动CCP1引脚为低电平

(3) 翻转CCP1引脚

(4) 无影响

(5) 使用硬件中断触发特殊事件,将定时器清零

可以使用CCP1CON寄存器来选择上述动作中的一种。如例15-1所示。注意,在匹配时,CCP1IF也将变为高电平。如图15-5所示。要使得上述的(1)、(2)和(3)能工作,必须将CCP引脚配置为输出引脚。从图15-4可以看出,使用T3CON寄存器来选择定时器1或定时器3用于比较模式。在同时带有CCP1和ECCP1模块的PIC18F452/458(或它们的新版本,PIC18F4520/4580)中,可以把定时器1分配给CCP1,把定时器3分配给ECCP1,这样就可以使它们相互独立地工作。注意,只有选项(5)(即特殊事件触发)才会将定时器1(或定时器3)清零,而对于其他任何情况都必须手动清零。

例15-1 根据图15-1和图15-4,请完成下面的工作。

(1) 如果希望在匹配时翻转CCP1引脚,请计算用于比较模式的CCP1CON寄存器的值。

(2) 如果希望将定时器3用于CCP1的比较模式,且不使用预分频器,请确定T3CON寄存器的值

解:

(1) 从图15-1可知,对于寄存器CCP1CON,有00000010B或0x20。

(2) 从图15-4可知,对于寄存器T3CON,有01000010B或0x42。

比较特征有很多的应用。其中一个应用就是统计穿过门的人数,当人数达到某个设定值时,将门关闭。

RD16	T3CCP2	T3CKPS1	T3CKPS0	T3CCP1	T3SYNC	TMR3CS	TMR3ON

RD16 D7 16位读写使能位

1 = 以单次16位的操作访问定时器3寄存器

0 = 以两次8位的操作访问定时器3寄存器

T3CCP2 : T3CCP1 D6位和D3位,将定时器3或定时器1分配给CCP1和CCP2模块

	CCP1	ECCP1(或CCP2)	
0 0 =	定时器1	定时器1	(用于比较/捕捉模式的时钟源)
0 1 =	定时器1	定时器3	(用于比较/捕捉模式的时钟源)
1 x =	定时器3	定时器3	(用于比较/捕捉模式的时钟源)

T3CKPS1 : T3CKPS0 D5位和D4位,定时器3输入时钟的预分频器值选择器

0 0 = 1:1 预分频器值

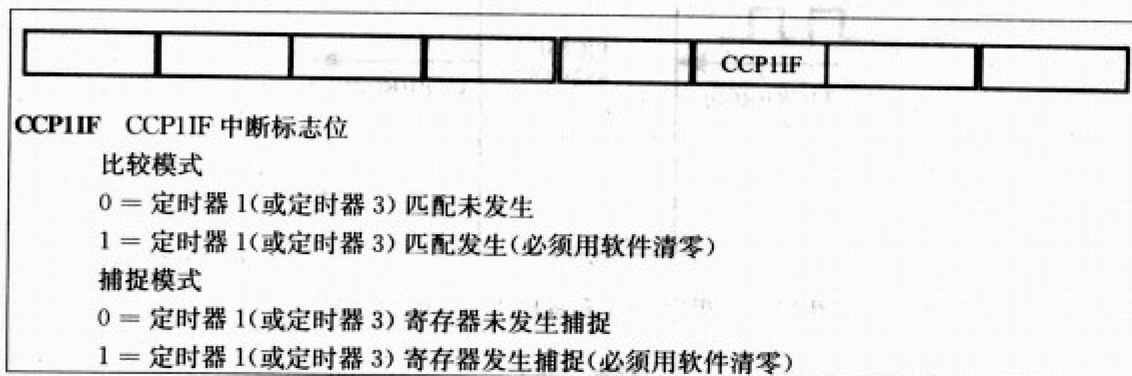
0 1 = 1:2 预分频器值

1 0 = 1:4 预分频器值

图15-4 T3CON(定时器3控制)寄存器

		1 1＝1∶8 预分频器值
T1SYNC	D2	定时器 3 外部时钟输入同步控制位 仅当 TMR3CS＝1、且时钟为外部时钟源时，该位可用。如果 TMR3CS＝0，则该位不可用 1＝不与外部输入时钟同步 0＝与外部输入时钟同步
TMR3CS	D1	定时器 3 时钟源选择位 1＝外部时钟源为 RC0(T1OSI 或者 T1CKI) 0＝内部时钟源($F_{osc}/4$)
TMR3ON	D0	定时器 3 开关控制位 1＝使能(启动)定时器 3 0＝停止定时器 3

图 15-4 (续)



CCP1IF　CCP1IF 中断标志位

比较模式

0＝定时器 1(或定时器 3)匹配未发生

1＝定时器 1(或定时器 3)匹配发生(必须用软件清零)

捕捉模式

0＝定时器 1(或定时器 3)寄存器未发生捕捉

1＝定时器 1(或定时器 3)寄存器发生捕捉(必须用软件清零)

图 15-5　PIR1(外围中断标志寄存器 1)包含 CCP1IF 标志

574

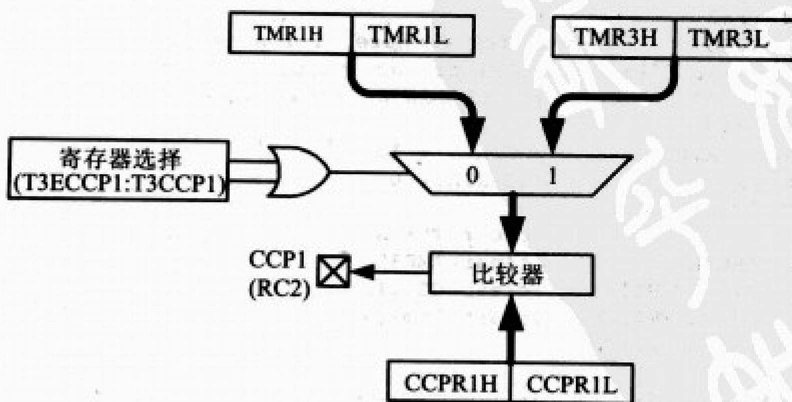


图 15-6　比较模式操作

15.2.1　比较模式编程的步骤

在编程比较模式时，必须遵循下列步骤。

(1) 初始化 CCP1CON 寄存器，用于比较模式。

(2) 为定时器1(或定时器3)初始化T3CON寄存器。

(3) 初始化CCPR1H：CCPR1L寄存器。

(4) 将CCP1引脚设为输出引脚。

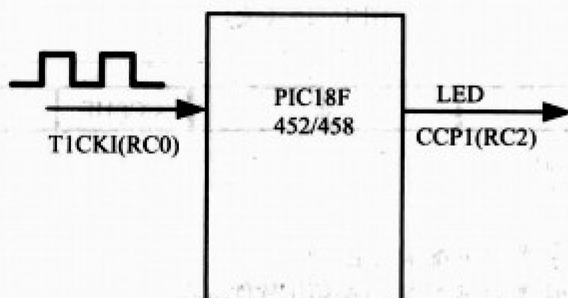
(5) 初始化定时器1(和定时器3)寄存器的值。

(6) 开启定时器1(和定时器3)。

(7) 监视CCP1IF标志位(或使用中断)。

程序15-1是比较模式的一个应用例子。它使用定时器3作为计数器，计数脉冲输入的个数。脉冲数可以是进入电梯的人数。当计数到10时，将翻转连接到CCP1引脚的LED。

对于程序15-1，假定1 Hz的脉冲连接到定时器3，LED连接到CCP1引脚。定时器3用作计数器。使用比较模式，该汇编程序将会每隔10个脉冲翻转LED一次。



575

图15-7 程序15-1和程序15-1C的示意图

程序 15-1

```
;Program 15-1
      MOVLW  0x02
      MOVWF  CCP1CON         ;Compare mode, toggle upon match
      MOVLW  0x42
      MOVWF  T3CON     ;Timer3 for Compare, 1:1 prescaler
      BCF    TRISC,CCP1      ;CCP1 pin as output
      BSF    TRISC,T1CKI     ;T3CLK pin as input pin
      MOVLW  D'10'
      MOVWF  CCPR1L          ;CCPR1L = 10
      MOVLW  0x0             ;CCPR1H = 0
      MOVWF  CCPR1H
OVER  CLRF   TMR3H           ;clear TMR3H
      CLRF   TMR3L           ;clear TMR3L
      BCF    PIR1,CCP1IF     ;clear CCP1IF
      BSF    T3CON,TMR3ON    ;start Timer3
B1    BTFSS  PIR1,CCP1IF
      BRA    B1
;---------CCP toggle CCP pin upon match
B2    BCF    T3CON,TMR3ON    ;stop Timer3
      GOTO   OVER            ;keep doing it

//Program 15-1C is a C version of Program 15-1
    CCP1CON=0x02;           //Compare mode, toggle upon match
    T3CON=0x42;       //Timer3 for Compare, 1:1 prescaler
    TRISCbits.TRISC2=0; //CCP1 pin an output
    TRISCbits.TRISC0=1;     //T3CLK pin an input
```

```
CCPR1L=10;              //load CCPR1L
CCPR1H=0;               //load CCPR1H
while(1)
{
  TMR3H=0;
  TMR3L=0;
  PIR1bits.CCP1IF=0;    //clear CCP1IF flag
  T3CONbits.TMR3ON=1;   //turn on Timer3
  while(PIR1bits.CCP1IF==0);//wait for CCP1IF
  //CCP toggles CCP pin upon match
  T3CONbits.TMR3ON=0;   //stop Timer3
}
```

考察程序 15-2：在该程序中，假定 PIC18452/458 的 F_{osc} = 10 MHz。它利用 CCP1 模块比较模式在 CCP1 引脚产生周期为 40 ms 的连续方波。方波的占空比为 50%，也就是说，每个周期中 CCP1 引脚有一半的时间处于高电平。这是一个将定时器 1 应用于比较模式的例子，如图 15-8 所示。定时器采用了 $F_{osc}/4$，因此时钟为 1/2.5 MHz = 0.4 μs。在 40 ms 的周期里，由 20 ms 高电平和 20 ms 低电平形成方波。20 ms/0.4 μs = 50 000 或 C350（十六进制），这就是要赋给 CCPR1H：CCPR1L 的值，以便使用比较模式。 576

RD16	—	T1CKPS1	T1CKPS0	T1OSCEN	T1SYNC	TMR1CS	TMR1ON

RD16　D7　16 位读写使能位
1 = 以单次 16 位的操作访问定时器 1 寄存器
0 = 以两次 8 位的操作访问定时器 1 寄存器

D6　未使用

T1CKPS2：T1CKPS0　D5 和 D4　定时器 1 的预分频器值选择器
0　0 = 1：1　预分频器值
0　1 = 1：2　预分频器值
1　0 = 1：4　预分频器值
1　1 = 1：8　预分频器值

T1OSCEN　D3　定时器 1 的振荡器使能位
1 = 使能定时器 1 的振荡器
0 = 关闭定时器 1 的振荡器

T1SYNC　D2　定时器 1 同步(仅当 TMR1CS = 1 时，在计数模式下同步外部时钟输入)
TMR1CS = 0，该位未使用

TMR1CS　D1　定时器 1 时钟源选择位
1 = 来自 RC0/T1CKI 引脚输入的外部时钟源
0 = 内部时钟源(来自 XTAL 的 $F_{osc}/4$)

TMR1ON　D0　定时器 1 开关控制位
1 = 使能(开启) 定时器 1
0 = 停止定时器 1

图 15-8　T1CON(定时器 1 控制) 寄存器

程序 15-2 使用比较模式在 CCP1 引脚上产生周期为 50ms、占空比为 50% 的方波。

程序 15-2

```
;Program 15-2
      MOVLW 0x02
      MOVWF CCP1CON  ;Compare mode, toggle upon match
      MOVLW 0x0
      MOVWF T3CON    ;use Timer1 for Compare mode
      MOVLW 0x0
      MOVWF T1CON    ;Timer1, internal CLK, 1:1 prescale
      BCF   TRISC,CCP1     ;CCP1 pin as output
      MOVLW 0xC3
      MOVWF CCPR1H   ;CCPR1H = 0xC3
      MOVLW 0x50

      MOVWF CCPR1L         ;CCPR1L = 0x50
OVER  CLRF  TMR1H          ;clear TMR1H
      CLRF  TMR1L          ;clear TMR1L
      BCF   PIR1,CCP1IF    ;clear CCP1IF
      BSF   T1CON,TMR1ON   ;start Timer1
B1    BTFSS PIR1,CCP1IF
      BRA   B1
      ;CCP toggles CCP1 pin upon match
      BCF   T1CON,TMR1ON   ;stop Timer1
      GOTO  OVER           ;keep doing it
```

577

程序 15-2C

```
//Program 15-2C is the C version of Program 15-2.
 CCP1CON=0x02;         //Compare mode, toggle upon match
 T3CON=0x0;            //Timer1 for Compare, 1:1 prescaler
 T1CON=0x0;            //Timer1 internal clk, 1:1 prescaler
 TRISCbits.TRISC2=0; //make CCP1 pin an output
 TRISCbits.TRISC0=1; //make T1CLK pin an input
 CCPR1H=0xC3;          //load CCPR1H
 CCPR1L=0x50;          //load CCPR1L
 while(1)
 {
   TMR1H=0;                      //clear Timer1
   TMR1L=0;
   PIR1bits.CCP1IF=0;            //clear CCP1IF flag
   T1CONbits.TMR1ON=1;           //turn on Timer1
   while(PIR1bits.CCP1IF==0);    //wait for CCP1IF
   //CCP toggles CCP1 pin upon match
   T1CONbits.TMR1ON=0;           //stop Timer1
 }
```

15.2.2 复习题

1. 判断对错:使用比较模式时,可选用任意的计时器。
2. 判断对错:比较模式有一个单独与之相关的引脚。
3. 判断对错:使用比较模式时,CCP 引脚必须配置为输入引脚。
4. 哪个寄存器用于选择比较模式的计时器?

578

15.3 捕捉模式编程

CCP1CON 寄存器的选择位可用来选择捕捉模式。在捕捉模式下，CCP 引脚上的事件会将定时器 1(或定时器 3) 的值赋给 16 位寄存器 CCPR1H：CCPR1L。因此，在捕捉模式下，必须将 CCP 引脚配置为输入。导致定时器 1(或定时器 3) 的内容被捕捉进 CCPR1H：CCPR1LD 寄存器的事件可以是由高到低的脉冲(下降沿)，也可以是由低到高的脉冲(上升沿)。就边沿触发脉冲而言，有 4 种方式可供选择：

(1) 每个下降沿脉冲

(2) 每个上升沿脉冲

(3) 每 4 个上升沿脉冲

(4) 每 16 个上升沿脉冲

通过 CCP1CON 的选择位，可以选择上面 4 种方式中的一种。如例 15-2 所示。注意，要使上面任意一种方式工作，必须将 CCP 引脚配置为输入引脚。

例 15-2 根据图 15-1 和图 15-4，请完成下面的要求。

(1) 如果希望在每个上升沿捕捉脉冲，请计算用于捕捉模式的 CCP1CON 寄存器的值。

(2) 如果希望将定时器 3 用于 CCP1 的捕捉模式，不使用预分频器，请计算 T3CON 寄存器的值。

解：

(1) 从图 15-1 可知，CCP1CON 寄存器的值为 00000101B 或 0x05。

(2) 从图 15-2 可知，T3CON 的值为 01000010B 或 0x42。

捕捉模式的一个应用是测量输入脉冲的频率。请参阅程序 15-3。

15.3.1 捕捉模式编程的步骤

在编程使用捕捉模式来测量脉冲周期时，必须遵循以下的步骤。

(1) 初始化 CCP1CON 寄存器，用于捕捉模式。

(2) 将 CCP1 引脚设为输入引脚。

(3) 初始化 T3CON 寄存器，以选择定时器 1 或定时器 3。

(4) 在第一个上升沿读定时器 1(或定时器 3) 的值，并保存。

(5) 在第二个上升沿读定时器 1(或定时器 3) 的值，并保存。

(6) 将第(5) 步所得的值减去第(4) 步所得的值。

15.3.2 测量脉冲周期

程序 15-3 是捕捉模式的一个应用例子。如图 15-9 所示。它测量由 CCP 引脚输入的脉冲周期，测量值为时钟周期的个数，即 $T_{sclk}[1/(F_{osc}/4)]$。如图 15-10 所示。

在程序 15-3 中，假定脉冲从 CCP 引脚输入。该汇编程序使用捕捉模式测量脉冲的周期数，并把结果放到 PORTB 和 PORTD 中。测量值是 $F_{osc}/4$ 时钟周期的个数，如图 15-11 所示。

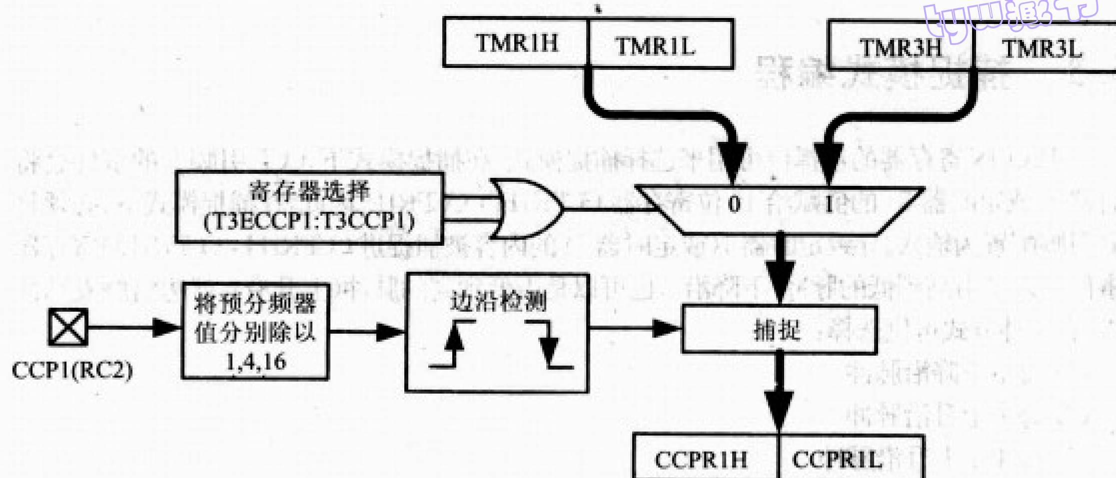


图 15-9　捕捉模式操作

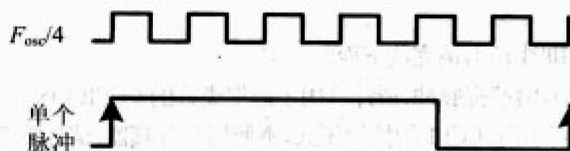
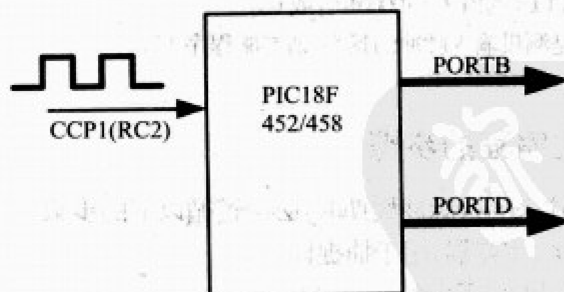


图 15-10　根据 $F_{osc}/4$ 时钟周期来测量脉冲周期



580

图 15-11　程序 15-3 和程序 15-3C 的示意图

程序　15-3

```
;Program 15-3
      MOVLW  0x05
      MOVWF  CCP1CON         ;Capture mode rising edge
      MOVLW  0x0
      MOVWF  T3CON           ;Timer1 for Capture
      MOVLW  0x0
      MOVWF  T1CON     ;Timer1, internal CLK, 1:1 prescale
      CLRF   TRISB           ;make PORTB output port
      CLRF   TRISD           ;make PORTD output port
      BSF    TRISC,CCP1      ;make CCP1 pin an input
      MOVLW  0x0
      MOVWF  CCPR1H          ;CCPR1H = 0
```

```
        MOVWF  CCPR1L          ;CCPR1L = 0
OVER    CLRF   TMR1H           ;clear TMR1H
        CLRF   TMR1L           ;clear TMR1L
        BCF    PIR1,CCP1IF     ;clear CCP1IF
RE_1    BTFSS  PIR1,CCP1IF
        BRA    RE_1            ;stay here for 1st rising edge
        BSF    T1CON,TMR1ON    ;start Timer1
        BCF    PIR1,CCP1IF     ;clear CCP1IF for next
RE_2    BTFSS  PIR1,CCP1IF
        BRA    RE_2            ;stay here for 2nd rising edge
        BCF    T1CON,TMR1ON    ;stop Timer1
        MOVFF  TMR1L,PORTB     ;put low byte on PORTB
        MOVFF  TMR1H,PORTD     ;put high byte on PORTD
        GOTO   OVER            ;keep doing it
```

程序 15-3C

```
//Program 15-3C is the C version of Program 15-3.
    CCP1CON=0x05;     //Capture mode on every rising edge
    T3CON=0x0;             //Timer1 for capture
    T1CON=0x0;        //Timer1 internal clk, 1:1 prescaler
    TRISB=0;               //make PORTB output port
    TRISD=0;               //make PORTD output port
    TRISCbits.TRISC2=1;    //make CCP1 pin an input
    CCPR1L=0;              //CCPR1L = 0
    CCPR1H=0;              //CCPR1H = 0
  while(1)
    {
    TMR1H=0;               //clear Timer1
    TMR1L=0;
    PIR1bits.CCP1IF=0;     //clear CCP1IF flag
    while(PIR1bits.CCP1IF==0);  //wait for 1st rising edge
    T1CONbits.TMR1ON=1;    //start Timer1
    PIR1bits.CCP1IF=0;     //clear CCPIF for next edge
    while(PIR1bits.CCP1IF==0);  //wait for 2nd rising edge
    T1CONbits.TMR1ON=0;    //stop Timer1
    PORTB=CCPR1L;
    PORTD=CCPR1H;          //display the clock count
    }
```

581

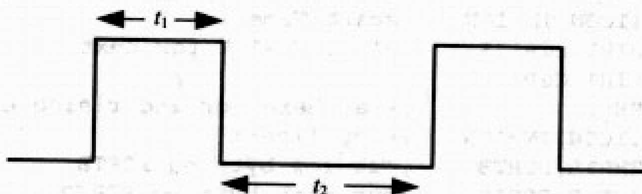
在上面程序中存在一个测量周期的问题，即由程序头引起的误差率，如果采用在第4或第16上升沿捕捉，可以减小程序头部的影响。

15.3.3 测量脉宽

测量脉宽是捕捉模式最广泛的应用之一。许多设备所测量的信号(如距离、温度等)，结果是以脉冲宽度的形式给出的，而不是传统的电压或电流形式。在这些设备中，输出为 PWM 格式。对于带有 PWM 输出的设备，输出都有固定的频率，可变的占空比就是要测量的数量。比如，Maxim 公司的 MAX6666/6667 温度传感器，“将周围环境的温度转换为定量 PWM 输出，温度信息包含在输出方波的占空比中。”根据它们的数据表，输出是在 +25℃ 时标称频率为 35 Hz 的方波。输出格式可解码为如下的方程：

$$温度(℃) = 235 - (400 \times t_1)/t_2 \quad (15\text{-}1)$$

其中，t_1 是固定的，典型值为 10 ms；t_2 是调制后的温度。在上面的公式中，$T = t_1 + t_2$，T 为脉冲周期，t_1 为脉冲高电平部分，t_2 为低电平部分，如图15-12所示。如果 $t_1 = 10$ ms，$t_2 = 20$ ms，那么温度 $= 235 - (400 \times 10)/20 = 235 - 200 = 35$。程序15-4描述了如何使用捕捉模式测量占空比。



582

图 15-12 Maxim 公司的 MAX6666/6667 温度传感器的占空比

程序 15-4

```
;Program 15-4
FLAG EQU 0x10    ;flag register for steps in detection
DISP EQU 0x0     ;flag for capture complete
RF    EQU 0x1    ;flag for rising or falling edge
      ORG 0x0000
      GOTO MAIN
      ORG 0x0008
      BTFSC PIR1,CCP1IF       ;Is it CCP1?
      GOTO CCP_ISR            ;service CCP1
      RETFIE                  ;else return
MAIN  MOVLW 0x05
      MOVWF CCP1CON           ;Capture mode rising edge
      MOVLW 0x0
      MOVWF T3CON             ;Timer1 for Capture
      MOVLW 0x0
      MOVWF T1CON ;Timer1, internal CLK, 1:1 prescale
      CLRF  TRISB             ;make PORTB output port
      CLRF  TRISD             ;make PORTD output port
      BSF   TRISC,CCP1        ;make CCP1 pin an input
      CLRF  CCPR1H            ;CCPR1H = 0
      CLRF  CCPR1L            ;CCPR1L = 0
      BSF   PIE1,CCP1IE       ;enable CCP1 interrupt
      BSF   INTCON,PEIE       ;enable peripheral interrupt
      BSF   INTCON,GIE        ;enable all interrupts
OVER  CLRF  TMR1H             ;clear TMR1H
      CLRF  TMR1L             ;clear TMR1L
WAIT  BTFSS FLAG,DISP         ;Is capture complete?
      BRA WAIT                ;else wait
      BCF FLAG,DISP           ;clear flag for next capture
      MOVLW 0x03
      SUBWF TMR1L,F           ;subtract the overhead
      MOVFF TMR1L,PORTB       ;put low byte on PORTB
      MOVFF TMR1H,PORTD       ;put high byte on PORTD
      GOTO  OVER              ;keep doing it
CCP_ISR    BTFSS FLAG,RF      ;Is it rising edge?
      GOTO RISE_ISR           ;service rising edge
      GOTO FALL_ISR           ;else service falling edge
RISE_ISR    BSF   T1CON,TMR1ON    ;start Timer1
      BSF FLAG,RF             ;ready for falling edge
      BCF CCP1CON,0           ;detect falling edge
```

```
      BCF PIR1,CCP1IF  ;clear interrupt
      RETFIE           ;return and wait for falling edge
FALL_ISR    BCF   T1CON,TMR1ON      ;stop Timer1
      BSF FLAG,DISP          ;capture complete
      BCF FLAG,RF            ;ready for rising edge
      BSF CCP1CON,0          ;detect rising edge
      BCF PIR1,CCP1IF  ;clear interrupt
      RETFIE           ;return capture complete
      END
```


程序 15-4C

```
//Program 15-4C
#include "p18f458.h"

void CCP1_ISR(void);
void rising(void);
void falling(void);
unsigned char disp = 0;
unsigned char rf = 0;

#pragma interrupt chk_isr
void chk_isr (void)
{
  if (PIR1bits.CCP1IF==1)
    CCP1_ISR();
}

#pragma code My_HiPrio_Int=0x0008
void My_HiPrio_Int (void)
{
  _asm
  GOTO chk_isr
  _endasm
}

#pragma code
void main()
  {
    CCP1CON=0x05;              //Capture mode rising edge
    T3CON=0x0;                 //Timer1 for Capture
    T1CON=0x0;                 //Timer1, internal CLK, 1:1 prescale
    TRISB=0x0;                 //make PORTB output port
    TRISD=0x0;                 //make PORTD output port
    TRISCbits.TRISC2=1;        //make CCP1 pin an input
    CCPR1H=0x0;                //CCPR1H = 0
    CCPR1L=0x0;                //CCPR1L = 0
       PIE1bits.CCP1IE=1;      //enable CCP1 interrupt
       INTCONbits.PEIE=1;      //enable peripheral interrupt
       INTCONbits.GIE=1;       //enable all interrupts
       while(1)
       {
      TMR1H=0x0;               //clear TMR1H
         TMR1L=0x0;            //clear TMR1L
      while(disp==0);          //Is data ready to display?
         disp=0;               //clear the flag
         TMR1L-=15;            //subtract the overhead
         PORTB=TMR1L;          //put low byte on PORTB
         PORTD=TMR1H;          //put high byte on PORTD
```

584

```
        }
      }
  void CCP1_ISR()
    {
      if(rf==0) rising();
         else falling();
      }

  void rising()
    {
       T1CONbits.TMR1ON=1;         //start Timer1
       rf=1;                       //ready for falling edge
       CCP1CONbits.CCP1M0=0;       //detect falling edge
       PIR1bits.CCP1IF=0;          //clear interrupt
    }

  void falling()
    {
       T1CONbits.TMR1ON=0;         //stop Timer1
       disp=1;                     //capture complete
       rf=0;                       //ready for rising edge
       CCP1CONbits.CCP1M0=1;       //detect rising edge
       PIR1bits.CCP1IF=0;          //clear interrupt
       }
```

注意,对于公司网站上的产品数据表,给定器件的输出可以是模拟信号(电压和电流),也可以是PWM。

15.3.4 复习题

1. 判断对错:在捕捉模式下,必须将CCP引脚配置为输入引脚。
2. 判断对错:对于捕捉模式,只能使用定时器1和定时器3。
3. 判断对错:每次CPU复位时,定时器的寄存器值就传送到CCPR1H:CCPR1L。
4. 判断对错:在定时器的寄存器值被传送到CCPR1H:CCPR1L后,定时器的寄存器将被清零。

585

5. 哪个寄存器用来选择定时器用于捕捉模式?

15.4 PWM编程

脉宽调制(PWM)是CCP的另一个重要特性。PWM特性允许产生可变脉宽的脉冲信号。尽管可以使用计时器来产生PWM,但使用CCP模块让PWM编程变得更容易、更有效率。PWM广泛地应用于工业控制中,如直流电机的控制,这将在第17章中见到。实际上,PWM的广泛应用使得Microchip公司为增强PWM容量,推出了新一代PIC18系列产品,即ECCP(增强型CCP)。ECCP将在下一节中介绍。ECCP与标准CCP的不同点在于PWM能力。在生成可变脉宽的脉冲时,有两个重要的因素:脉冲周期和占空比。占空比(duty cycle,DC)指的是脉冲周期中高电平部分所占的比例,通常以百分数的形式给出。例如,一个周期为4 ms的脉冲,其中1 ms是高电平,则占空比为25%(1 ms/4 ms = 25%),如图15-13所示。

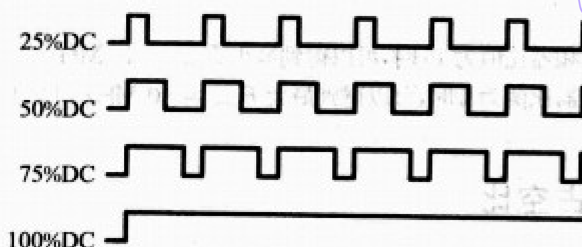


图 15-13 周期与占空比

15.4.1 PWM 周期

CCP 模块使用定时器 2 和与之有关的寄存器(PR2)来作为 PWM 时间基准，也就是说，PWM 的频率是晶振频率(F_{osc})的一部分。使用 PR2 来设置 PWM 周期，如下所示：

$$T_{pwm} = [(PR2)+1]4 \times N \times T_{osc} \tag{15-2}$$

其中，T_{osc} 是晶振频率的倒数，即 $1/F_{osc}$，T_{pwm} 是期望的 PWM 周期，N 为预分频器值(1、4 或 16)，这由 T2CON 寄存器来设置。因此，可得到计算 PR2 寄存器值的公式如下：

$$PR2 = [F_{osc}/(F_{pwm} \times 4 \times N)] - 1 \tag{15-3}$$

由方程(15-2)可推知，当 $N = 16$、$PR2 = 255$ 时，T_{pwm} 取得最大值，即：

$$T_{pwm} = [(255)+1] \times 4 \times 16 \times T_{osc} = 16\ 382 T_{osc}$$

也就是说，F_{pwm} 的最小取值为 $F_{osc}/16\ 382T_{osc}$。

586

考察例 15-3 ～ 例 15-5，学习如何计算 PWM 的周期。

例 15-3 如果要得到下面的 PWM 频率，请计算 PR2 寄存器的值和预分频器值。假定 XTAL = 20 MHz。

(a) 1.22 kHz；(b) 4.88 kHz；(c) 78.125 kHz

解：

(a) PR2 值 = [(20 MHz/(4×1.22 kHz)] − 1 = 4097，这大于 PR2 的最大允许值 255。因此，现在选择预分频器比值为 16，则 PR2 值 = [(20 MHz/(4×1.22 kHz×16)] − 1 = 255。

(b) PR2 值 = [(20 MHz/(4×4.88 kHz)] − 1 = 1023，这大于 PR2 的最大允许值 255。因此，现在选择预分频器比值为 4，则 PR2 值 = [(20 MHz/(4×4.88 kHz×4)] − 1 = 255。

(c) PR2 值 = [(20 MHz/(4×78.125 kHz)] − 1 = 63。

例 15-4 如果要得到下面的 PWM 频率，请计算 PR2 的值。假定 XTAL = 10 MHz，预分频器比值为 1。

(a) 10 kHz；(b) 25 kHz

解：

(a) PR2 值 = [(10 MHz/(4×10 kHz×1)] − 1 = 250 − 1 = 249

(b) PR2 值 = [(10 MHz/(4×25 kHz×1)] − 1 = 100 − 1 = 99

例 15-5 请计算在 XTAL = 10 MHz 时 F_{pwm} 频率的最小值和最大值。针对 F_{pwm} 频率的最小值和最大值，分别计算 PR2 的值和预分频器值。

解:

当 PR2 = 255 且预分频器比值为 16 时,可以得到最小 F_{pwm} = 10 MHz/(4×16×256) = 610Hz。

587

当 PR2 = 1 且预分频器比值为 1 时,可以得到最大 F_{pwm} = 10 MHz/(4×1×1) = 2.5 MHz。

15.4.2 PWM 的占空比

正如前面提到的,PWM 的占空比指的是脉冲周期中高电平所占的比例。为设置占空比,CCP 模块使用了 10 位寄存器 DC1B9 : DC1B0。该 10 位寄存器 DC1B9 : DC1B0 由 CCPR1L 的 8 位和 CCP1CON 的 2 位组成,其中 CCPR1L 为高 8 位,CCP1CON 的 DC1B2 : DC1B1 为低 2 位。实际上,CCPR1L 是占空比的主要寄存器,低 2 位 DC1B2 : DC1B1 是占空比的小数部分,其设置如下:

DC1B2	DC1B1	十进制小数
0	0	0
0	1	0.25
1	0	0.5
1	1	0.75

注意,占空比寄存器 CCPR1L 的值始终是 PR2 的一个百分数。例如,如果 PR2 = 50,需要的占空比为 20%,那么 CCPR1L = 10,因为 20% × 50 = 10。在这种情况下,DC1B2 : DC1B1 = 00。现在假定想要 25% 的占空比,PR2 保持不变,由于 50 × 25% = 12.5,所以需要设 CCPR1L = 12,DC1B2 : DC1B1 = 10。请参阅例 15-6 做进一步的了解。

例 15-6 如果 PWM 有如下的频率,占空比为 75%,请计算 PR2、CCP1RL 和 DC1B2 : DC1B1 的值。假定 XTAL = 10 MHz。

(a) 1 kHz; (b) 2.5 kHz

解:

(a) 根据方程 PR2 = F_{osc}/(4 × F_{pwm} × N),必须设预分频器比值 N = 16。则:

PR2 = [(10 MHz/(4 × 1 kHz × 16)] − 1 = 156 − 1 = 155

因为 155 × 75% = 116.25,所以 CCPR1L = 116,DC1B2 : DC1B1 = 01(用于 0.25 部分)

(b) 根据方程 PR2 = F_{osc}/(4 × F_{pwm} × N),可以设预分频器比值 N = 4。则:

PR2 = [(10 MHz/(4 × 1 kHz × 4)] − 1 = 250 − 1 = 249

588

因为 249 × 75% = 186.75,所以 CCPR1L = 186,DC1B2 : DC1B1 = 11(用于 0.75 部分)

15.4.3 PWM 编程的步骤

在编程 CCP 模块的 PWM 特征时,其编程步骤如下。

(1) 写 PR2 寄存器,设置 PWM 周期。

(2) 写 CCPR1L 寄存器,设置 PWM 占空比的高 8 位。

(3) 将 CCP 引脚设置为输出。

(4) 使用 T2CON 设置预分频器值,如图 15-14 所示。

(5) 将 TMR2 清零。

(6) 为 PWM 配置 CCP1CON 寄存器;为占空比的小数部分设置 DC1B2 : DC1B1。

(7) 启用定时器 2。

TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0

D6 D0

D7 未使用

TOUTPS3 : TOUTPS0 D6—D3 定时器 2 输出的后分频器选择位

00 0 0 = 1 : 1 后分频器值

00 0 1 = 1 : 2 后分频器值

00 1 0 = 1 : 3 后分频器值

00 1 1 = 1 : 4 后分频器值

11 1 0 = 1 : 15 后分频器值

11 1 1 = 1 : 16 后分频器值

TMR2ON D2 定时器 2 开关控制位

1 = 开启定时器 2

0 = 停止定时器 2

T2CKPS1 : T2CKPS0 D1—D0 定时器 2 时钟预分频器选择位

0 0 = 预分频器比为 1

0 1 = 预分频器比为 4

1 x = 预分频器比为 16

图 15-14 T2CON(定时器 2 控制) 寄存器

研究程序 15-5 和程序 15-5C,观察如何编程 PWM 特性。这些程序用了 TMR2IF 标志位。如图 15-15 所示。

						TMR2IF	TMR1IF

TMR2IF 定时器 2 中断溢出标志位

0 = TMR2 的值不等于 PR2 寄存器的值

1 = TMR2 的值等于 PR2 寄存器的值

TMRxIF 在 PIR 中的位置在未来的产品中可能会不同。

图 15-15 PIR1(外围中断标志寄存器 1) 的 TMR2IF 标志位

589

使用例 15-6 的数据,程序 15-5 在 CCP1 引脚上产生 2.5 kHz 的 PWM,占空比为 75%。

程序 15-5

```
;Program 15-5
      CLRF    CCP1CON          ;clear CCP1CON reg
      MOVLW   D'249'
      MOVWF   PR2
      MOVLW   D'186'           ;75% duty cycle
      MOVWF   CCPR1L
      BCF     TRISC,CCP1       ;make PWM pin an output
      MOVLW   0x01       ;Timer2, 4 prescale, no postscaler
      MOVWF   T2CON
      MOVLW   0x3C             ;PWM mode, 11 for DC1B1:B0
      MOVWF   CCP1CON
      CLRF    TMR2             ;clear Timer2
      BSF     T2CON,TMR2ON     ;turn on Timer2
 AGAIN BCF    PIR1,TMR2IF      ;clear Timer2 flag
 OVER BTFSS   PIR1,TMR2IF      ;wait for end of period
```

```
    BRA    OVER
    GOTO   AGAIN             ;continue
```

程序　15-5C

```
//Program 15-5C is the C version of Program 15-5.
 CCP1CON=0;                  //clear CCP1CON reg
 PR2=249;
 CCPR1L=186;                 //75% duty cycle
 TRISCbits.TRISC2=0;         //make PWM pin an output
 T2CON=0x01;            //Timer2, 4 prescale, no postscaler
 CCP1CON=0x3C;               //PWM mode, 11 for DC1B1:B0
 TMR2=0;                     //clear Timer2
 T2CONbits.TMR2ON=1;         //turn on Timer2
 while(1)
 {
   PIR1bits.TMR2IF=0;        //clear Timer2 flag
   while(PIR1bits.TMR2IF==0); //wait for end of period
 }
```

必须注意 CCPR1H 在产生占空比的过程中所起的作用。只要开启定时器 2,CCPR1L 里的占空比就被复制到 CCPR1H。产生 PWM 时,定时器 2 要经历下面的几个阶段。

(1) 将 CCPR1L 的值送到 CCPR1H,CCP1 引脚变为高电平,PWM 周期开始。

(2)TMR2 开始计数,将其值与 CCPR1H 和 PR2 寄存器的值做比较。

(3) 当 TMR2 与 CCPR1H(与 CCPR1L 相等) 的值相等时,CCP 引脚变为低电平。结束占空比部分。

(4)TMR2 继续计数,直到其值等于 PR2。此时 CCP 引脚变为高电平,表明 PWM 周期结束,同时开始下一个 PWM 周期。将定时器 2 清零,为下一个周期做准备。将 CCPR1L 的值送到 CCPR1H,继续下一个过程。如图 15-16 和图 15-17 所示。

590

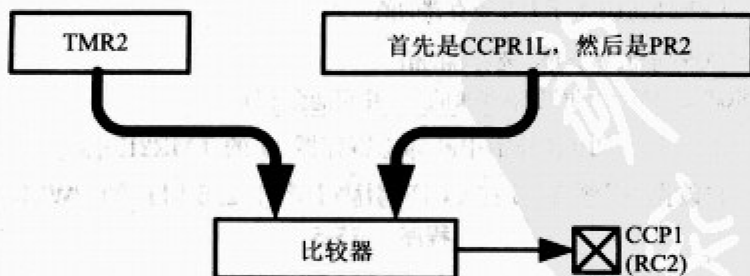


图 15-16　在占空比产生过程中 TMR2 和 PR2 的作用

注意,由于CCPR1L是PR2的一部分,定时器2先匹配CCPR1L,然后才匹配PR2,除非占空比为 100%。如果占空比为 100%,那么定时器 2 同时与 CCPR1L 和 PR2 相匹配,因为它们的值相等。

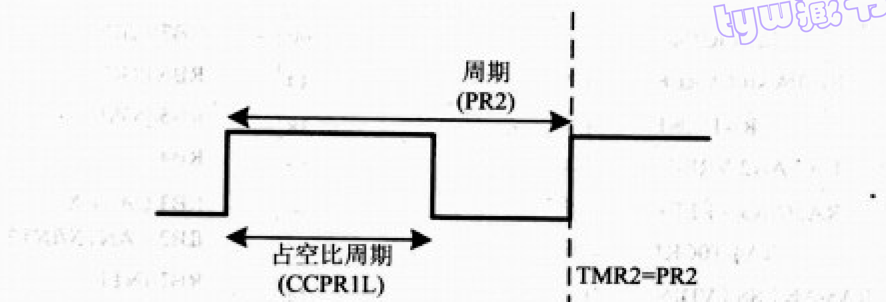


图 15-17 PWM 中关于 CCPR1L 和 PR2 的 TMR2

15.4.4 占空比与 F_{osc}

PIC18 的数据表给出了 F_{osc} 与占空比周期之间的关系：

$$T_{dutycycle} = (\text{DC1B9} : \text{DC1B0 值}) \times T_{osc} \times N \tag{15-4}$$

其中，$T_{osc} = 1/F_{osc}$，N 为预分频器比值(1、4 或 16)，这由定时器 2 的控制寄存器来设置。为得到 DC1B9：DC1B0 的值，重新整理上式：

$$\text{DC1B9} : \text{DC1B0} = [F_{osc}/(F_{dutycycle} \times N)] \tag{15-5}$$

为计算 PWM 的最大分辨率(位数)，PIC 用户手册给出了下面的方程：

最大的 PWM 分辨率(位) = $\log(F_{osc}/F_{pwm})/\log(2)$ 位

591

注意最大分辨率是 10 位。

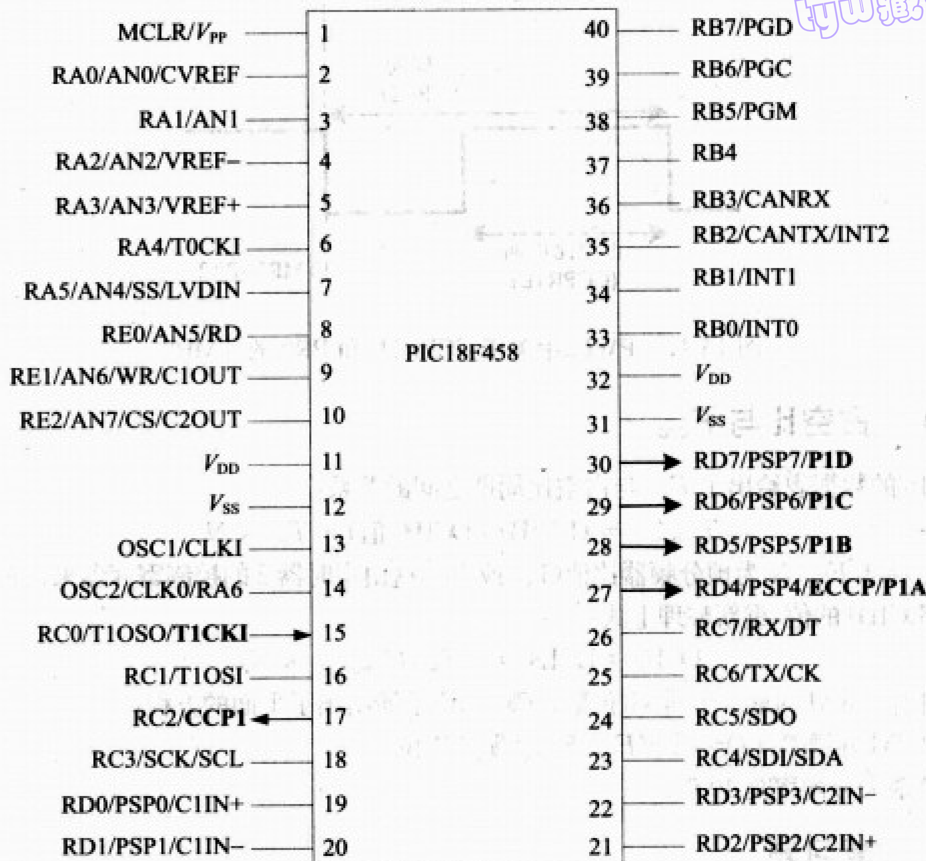
15.4.5 复习题

1. 判断对错：每个标准 CCP 模块都只有一个 PWM 引脚。
2. PIC18F458/4580 有多少个标准 CCP 模块?
3. 判断对错：对于 CCP1，必须使用 PR2 来设置 PWM 周期。
4. 判断对错：对于 CCP1，必须使用 CCPR1L 来设置 PWM 占空比。
5. PIC18F458/4580 的哪个引脚用于 PWM?
6. 判断对错：占空比总是周期的一部分，除非占空比为 100%。

15.5 ECCP 编程

许多 PIC18F 系列芯片除了带有标准 CCP 外，还带有 ECCP(增强型 CCP)。标准 CCP 模块称为 CCP1、CCP2 等，而 ECCP 模块也被命名为 ECCP1、ECCP2 等。如同标准 CCP 一样，ECCP 也有自己的引脚和寄存器。PIC18F452/458 使用 RD4(PORTD.4) 作为 ECCP1 引脚，使用 RC2(PORTC.2) 作为标准 CCP 引脚。如图 15-18 所示。图 15-19 给出了 ECCP1 控制寄存器。

ECCP1 包含 ECCPR1L 寄存器、ECCPR1H 寄存器和 ECCPCON1 寄存器。ECCP1IF 标志位于 PIR2 寄存器内。如图 15-20 和图 15-21 所示。同标准 CCP 一样，可以使用定时器 1、定时器 2 和定时器 3 对比较 - 捕捉和 PWM 特征进行编程。如表 15-3 所示。



592

图 15-18 PIC18F458/4580(452/4520) 用于 PWM 的 ECCP 引脚

EPWM1M1	EPWM1M0	EDC1B1	EDC1B0	ECCP1M3	ECCP1M2	ECCP1M1	ECCP1M0
D7							D0

EPWM1M1：EPWM1M0 PWM 输出引脚配置。它允许一个引脚用于捕捉 / 比较模式，或者 4 个引脚用于 PWM。在捕捉 / 比较模式下，只使用了 P1A(RD4) 引脚，此时，这两位没有选择功能。在 PWM 模式下，这两位的选择功能如下：

00 P1A 用于调制输出。P1B、P1C 和 P1D 作为 I/O 端口
01 全桥正向输出。P1D 调制输出，P1A 有效，P1B 和 P1C 无效
10 半桥输出。P1A 和 P1B 带死区控制调制输出，P1C 与 P1D 作为 I/O 端口
11 全桥反向输出。P1B 调制输出，P1C 有效，P1A 和 P1D 无效

EDC1B 10：EDC1B1 PWM 占空比的最低有效位，仅用于 PWM。10 位占空比寄存器的最低有效位(第 1 位和第 0 位) 用于 PWM，而 ECCPR1L 寄存器用作 10 位占空比寄存器的第 2 位至第 9 位

ECCP1M3 ～ ECCP1M0 ECCP1 模式选择

0000 ECCP1 关闭

图 15-19 ECCP1 控制寄存器(该寄存器用来选择捕捉、比较操作模式中的一种或 EECP1 的 PWM)

0001	保留
0010	比较模式,在匹配时翻转 ECCP1 输出引脚(ECCP1IF 置位)
0011	保留
0100	捕捉模式,在每个上升沿发生捕捉
0101	捕捉模式,在每个下降沿发生捕捉
0110	捕捉模式,每 4 个上升沿发生捕捉
0111	捕捉模式,每 16 个上升沿发生捕捉
1000	比较模式。初始化 ECCP1 引脚为低电平,在比较匹配时,将其置为高电平(ECCP1IF 置位)
1001	比较模式。初始化 ECCP1 引脚为高电平,在比较匹配时,将其置为低电平(ECCP1IF 置位)
1010	比较模式。在比较匹配时产生软件中断(ECCP1IF 置位,ECCP1 引脚无变化)
1011	比较模式。触发特殊事件(ECCP1IF 置位,定时器 1 或定时器 3 复位为 0)
1100	PWM 模式。P1A、P1C 高电平有效;P1B 和 P1D 高电平有效
1101	PWM 模式。P1A、P1C 高电平有效;P1B 和 P1D 低电平有效
1110	PWM 模式。P1A、P1C 低电平有效;P1B 和 P1D 高电平有效
1111	PWM 模式。P1A、P1C 低电平有效;P1B 和 P1D 低电平有效

图 15-19 (续)

593



图 15-20 ECCP 的高、低字节寄存器

							ECCP1IF

ECCP1IF ECCP1IF 中断标志位

比较模式

0 = 定时器 1(或定时器 3) 匹配没发生

1 = 定时器 1(或定时器 3) 匹配发生(必须用软件清零)

捕捉模式

0 = 定时器 1(或定时器 3) 寄存器捕捉没发生

1 = 定时器 1(或定时器 3) 寄存器捕捉发生(必须用软件清零)

图 15-21 PIR2(外围中断标志寄存器 2) 包含 ECCP1IF 标志位

表 15-3 PIC18 的 ECCP1 定时器的使用

ECCP 模式	定时器
捕捉	定时器 1(或定时器 3)
比较	定时器 1(或定时器 3)
PWM	定时器 2

15.5.1 ECCP 比较模式的编程步骤

除了使用的是 ECCP 寄存器外,ECCP 的比较模式编程与标准 CCP 一样,其步骤如下。

(1) 初始化 ECCP1CON 寄存器,用于比较模式。

(2) 初始化 T3CON 寄存器,用于设置定时器 1(或定时器 3)。

(3) 初始化寄存器 ECCPR1H：ECCPR1L。

(4) 设 ECCP1 引脚为输出引脚。

(5) 初始化定时器 1(或定时器 3) 寄存器值。

(6) 开启定时器 1(或定时器 3)。

594

(7) 监视 ECCP1IF 标志位(或使用中断)。

程序 15-6 是比较模式的一个应用例子。定时器 3 用作计数器,计数输入定时器 3 的脉冲。当计数值到达 20 时,将翻转连接到 ECCP1 引脚的 LED。

对于程序 15-6,假定连接到定时器 3 引脚的脉冲频率为 1 Hz,LED 与 CCP1 引脚相连。定时器 3 用作计数器。使用比较模式,下面的汇编程序每隔 20 个脉冲将 LED 翻转一次。

程序　15-6

```
;Program 15-6
        MOVLW  0x02
        MOVWF  ECCP1CON  ;Compare mode, toggle upon match
        MOVLW  0x42
        MOVWF  T3CON     ;Timer3 for Compare, 1:1 prescaler
        BCF    TRISD,ECCP1    ;ECCP pin as output
        BSF    TRISC,T1CKI    ;T3CLK pin as input pin
        MOVLW  D'20'
        MOVWF  ECCPR1L        ;ECCPR1L = 20
        MOVLW  0x0            ;ECCPR1H = 0
        MOVWF  ECCPR1H
 OVER CLRF     TMR3H          ;clear TMR3H
        CLRF   TMR3L          ;clear TMR3L
        BCF    PIR2,ECCP1IF   ;clear ECCP1IF
        BSF    T3CON,TMR3ON   ;start Timer3
 B1     BTFSS  PIR2,ECCP1IF
        BRA    B1
 ;---------CCP toggle CCP pin upon match
 B2     BCF    T3CON,TMR3ON   ;stop Timer3
        GOTO   OVER           ;keep doing it
```

程序　15-6C

```
//Program 15-6C is the C version of Program 15-6.
      ECCP1CON=0x02;  //Compare mode, toggle upon match
      T3CON=0x42;     //Timer3 for Compare, 1:1 prescaler
      TRISDbits.TRISD4=0;    //make ECCP1 pin an output
      TRISCbits.TRISC0=1;    //make T3CLK pin an input
      ECCPR1L=20;            //load ECCPR1L
      ECCPR1H=0;             //load ECCPR1H
      while(1)
      {
        TMR3H=0;
        TMR3L=0;
        PIR2bits.ECCP1IF=0;            //clear ECCP1IF flag
        T3CONbits.TMR3ON=1;            //turn on Timer3
        while(PIR2bits.ECCP1IF==0);    //wait for CECP1IF
        //ECCP toggles ECCP pin upon match
```

```
    T3CONbits.TMR3ON=0;              //stop Timer3
  }
```


595

15.5.2 ECCP 捕捉模式的编程步骤

除了使用的是 ECCP 寄存器外，ECCP1 的捕捉模式编程与标准 CCP 相同。使用 ECCP1 捕捉模式来测量脉冲周期的步骤如下。

(1) 初始化 ECCP1CON 寄存器，用于捕捉模式。

(2) 设 ECCP1 引脚为输入引脚。

(3) 初始化 T3CON 寄存器，来选择定时器 1 或定时器 3。

(4) 在第一个上升沿读定时器 1(或定时器 3) 寄存器值并保存。

(5) 在第二个上升沿读定时器 1(或定时器 3) 寄存器值并保存。

(6) 将第(5) 步所得值减去第(4) 步所得值。

对于程序 15-7，假定脉冲由 ECCP1 引脚输入。下面的汇编程序使用捕捉模式测量脉冲周期，并将其放到 PORTB 和 PORC 中。测量值是 $F_{osc}/4$ 时钟周期的个数。

程序 15-7

```
;Program 15-7
        MOVLW   0x05
        MOVWF   ECCP1CON        ;Capture mode on rising edge
        MOVLW   0x0
        MOVWF   T3CON           ;Timer1 for capture
        MOVLW   0x0
        MOVWF   T1CON       ;Timer1, internal clk, 1:1 prescale
        CLRF    TRISB           ;make PORTB output port
        CLRF    TRISC           ;make PORTC output port
        BSF     TRISD,ECCP1     ;make ECCP1 pin an input
        MOVLW   0x0
        MOVWF   CCPR1H          ;ECCPR1H = 0
        MOVWF   CCPR1L          ;ECCPR1L = 0
OVER    CLRF    TMR1H           ;clear TMR1H
        CLRF    TMR1L           ;clear TMR1L
        BCF     PIR2,ECCP1IF    ;clear ECCP1IF
RE_1    BTFSS   PIR2,ECCP1IF
        BRA     RE_1            ;stay here for 1st rising edge
        BSF     T1CON,TMR1ON    ;start Timer1
        BCF     PIR2,ECCP1IF    ;clear ECCP1IF for next
RE_2    BTFSS   PIR2,ECCP1IF
        BRA     RE_2            ;stay here for 2nd rising edge
        BCF     T1CON,TMR1ON    ;stop Timer1
        MOVFF   TMR1L,PORTC     ;put low byte on PORTC
        MOVFF   TMR1H,PORTD     ;put high byte on PORTD
        GOTO    OVER            ;keep doing it
```

程序 15-7C

596

```
//Program 15-7C is the C version of Program 15-7.
    ECCP1CON=0x05;    //Capture mode on every rising edge
    T3CON=0x0;              //Timer1 for capture
    T1CON=0x0;        //Timer1, internal clk, 1:1 prescaler
    TRISC=0;                //make PORTB output port
```

```
    TRISD=0;                //make PORTD output port
    TRISDbits.TRISD4=1;     //make ECCP1 pin an input
    ECCPR1L=0;              //ECCPR1L = 0
    ECCPR1H=0;              //ECCPR1H = 0
  while(1)
    {
    TMR1H=0;                //clear Timer1
    TMR1L=0;
    PIR2bits.ECCP1IF=0;     //clear ECCP1IF flag
    while(PIR2bits.ECCP1IF==0);  //wait for 1st rising edge
    T1CONbits.TMR1ON=1;     //start Timer1
    PIR2bits.ECCP1IF=0;     //clear ECCPIF for next edge
    while(PIR2bits.ECCP1IF==0);  //wait for 2nd rising edge
    T1CONbits.TMR1ON=0;     //stop Timer1
    PORTC=CCPR1L;
    PORTD=CCPR1H;           //display the clock count
    }
```

15.5.3 ECCP 的 PWM 特征

ECCP同标准CCP模块之间的主要不同在于PWM能力。标准CCP只允许单个PWM输出引脚,这对于广泛用于直流电机控制的半桥实现是不够的。在第17章将会看到,需要4个引脚来驱动半桥,实现直流电机控制。ECCP允许使用4个引脚实现全桥控制和使用2个引脚实现半桥控制。ECCP用到的4个引脚如表15-4所示。关于占空比的计算,ECCP1与CCP1相同,使用PR2寄存器来设置占空比。

表 15-4 PIC18 用于 ECCP1 的引脚用途

ECCP 模式	RD4	RD5	RD6	RD7
比较/捕捉	ECCP1	I/O	I/O	I/O
双输出 PWM	P1A	P1B	I/O	I/O
四输出 PWM	P1A	P1B	P1C	P1D

注意:I/O是指它们用于输入/输出,或者与这些引脚有关的其他功能。

15.5.4 ECCP 的 PWM 编程步骤

对ECCP模块的PWM特征编程,可遵循下面的步骤。

(1) 通过写PR2寄存器设置PWM周期。

597

(2) 通过写ECCPR1L设置PWM占空比高8位。

(3) 设置ECCP引脚为输出。

(4) 使用T2CON设置预分频器。

(5) 将TMR2寄存器清空。

(6) 为PWM配置ECCP1CON寄存器,为占空比的小数部分设置EDC1B2:EDC1B1。

(7) 开启定时器2。

注意,在比较/捕捉特征的编程过程中,可以将定时器1分配给标准CCP1,将定时器3分配给ECCP1(或者反过来)。然而对于PWM,只有一个寄存器用来设置占空比。因此,如果对CCP1和ECCP1进行PWM特征编程,它们将会有同样的周期,因为只有一个PR2寄存器来设

置周期。在第 17 章将介绍如何使用 ECCP 来实现 4 引脚半桥的直流电机控制。

15.5.5 复习题

1. 判断对错:每个 ECCP 模块只有 1 个引脚用于 PWM。
2. PIC18F458/4580 有多少个 ECCP 模块?
3. 判断对错:对于 ECCP1,必须使用 PR2 寄存器来设置 PWM 周期。
4. 判断对错:对于 ECCP1,必须使用 CCPR1L 寄存器来设置 PWM 占空比。
5. PIC18F458/4580 的哪些引脚用于 PWM?

小结

本章以描述 PIC18 系列的 CCP 特征开始,然后讨论了标准 CCP 和增强型 CCP(ECCP) 模块,分别介绍了它们的比较、捕捉和 PWM 特征。本章还介绍了怎样使用定时器 1 或定时器 3 作为比较和捕捉模式的时间基准,以及使用定时器 2 实现 PWM 脉宽调制。

习题

1. 判断对错:PIC18 系列每款芯片都有片上 CCP 模块。
2. 判断对错:PIC18F452/458 只有一个标准 CCP。
3. 判断对错:PIC18F452/458 只有一个 ECCP 模块。
4. 判断对错:每个 CCP 模块有一个 16 位寄存器,可通过 CCPRL 和 CCPRH 访问。
5. 判断对错:每个 CCP 模块有一个单独的引脚。
6. 指出 PIC18F4520/4580 标准 CCP 和增强型 CCP 模块的数目。
7. 指出在 PIC18F4520/4580 40 引脚 DIP 封装中用于标准 CCP 的引脚。
8. 判断对错:可以使用 CCP1CON 来选择比较模式。
9. 判断对错:可以将定时器 0 和定时器 2 用于比较模式。
10. 判断对错:为使用比较模式,必须将 CCP 引脚设置为输出引脚。
11. 哪些定时器可用于比较模式?
12. 假定选择定时器 1 用于比较模式,指出何时 CCP 引脚会变为高电平?
13. 哪个寄存器包含 CCP 标志位?
14. 如果在比较匹配时 CCP 引脚变为高电平,请计算 CCP1CON 寄存器的值。
15. 如果在比较匹配时 CCP 引脚变为低电平,请计算 CCP1CON 寄存器的值。
16. 如果在比较匹配时 CCP 引脚电平反转,请计算 CCP1CON 寄存器的值。
17. 重写程序 15-1(15-1C),要求使用定时器 1。
18. 重写程序 15-1(15-1C),要求计数值为 1000。
19. 重写程序 15-2(15-2C),要求使用定时器 3。
20. 重写程序 15-2(15-2C),要求产生频率为 100 Hz 的方波。
21. 判断对错:可以使用 CCP1CON 来选择捕捉模式。
22. 判断对错:可以将定时器 0 和定时器 2 用于捕捉模式。
23. 判断对错:为使用捕捉模式,必须将 CCP 引脚设为输出引脚。
24. 哪些计时器可用于捕捉模式?

598

25. 如果在下降沿发生捕捉，请计算 CCP1CON 寄存器的值。
26. 如果每 4 个上升沿发生捕捉，请计算 CCP1CON 寄存器的值。
27. 如果定时器 1 用于捕捉模式，请计算 T3CON 寄存器的值。
28. 重写程序 15-3(15-3C)，要求使用定时器 3。
29. 判断对错：可以使用 CCP1CON 来选择 PWM 模式。
30. 判断对错：可以将定时器 0 和定时器 2 用于 PWM 模式。
31. 判断对错：为使用 PWM 模式，必须将 CCP 引脚设为输出引脚。
32. 对于标准 CCP1，哪个定时器用于 PWM 模式？
33. 请计算 PWM 模式下 CCP1CON 寄存器的值。
34. 在 CCPR1L 和 CCPR1H 寄存器中，哪个寄存器用来设置占空比？
35. 哪个寄存器包含 DC1B2：DC1B1？
36. 在设置占空比时，DC1B2：DC1B1 的作用是什么？

599

37. 如果占空比的小数部分是 0.75，那么 DC1B2：DC1B1 的值是什么？
38. 在 PWM 编程时，赋值给 CCPRL1 的值通常是 PR2 的_______(分数，倍数)。
39. 假定 PWM 频率为 2 kHz，25% 占空比，XTAL = 10 MHz。请计算 PR2、CCP1RL 和 DCB1B2：DC1B1 的值。
40. 假定 PWM 频率为 1.8 kHz，25% 占空比，XTAL = 10 MHz。请计算 PR2、CCP1RL 和 DCB1B2：DC1B1 的值。
41. 假定 PWM 频率为 1.5 kHz，25% 占空比，XTAL = 10 MHz。请计算 PR2、CCP1RL 和 DCB1B2：DC1B1 的值。
42. 假定 PWM 频率为 1.2 kHz，25% 占空比，XTAL = 10 MHz。请计算 PR2、CCP1RL 和 DCB1B2：DC1B1 的值。
43. 判断对错：可以使用 ECCP1CON 来选择 PWM 模式。
44. 判断对错：在 ECCP 下，可以将定时器 1 和定时器 3 用于 PWM 模式。
45. 判断对错：为使用捕捉模式，必须将 ECCP 引脚设为输出引脚。
46. 对于 ECCP1，哪个定时器用于 PWM 模式？
47. 哪个寄存器包含 ECCP1IF 标志位？
48. 如果在比较匹配时 ECCP 引脚变为高电平，请计算 ECCP1CON 寄存器的值。
49. 找出 PIC18F452/458 芯片用于 ECCP 捕捉／比较模式的引脚。
50. 哪些定时器可用于 ECCP 比较模式？
51. 在 ECCP1 下，PWM 模式使用了哪些引脚？
52. 如果在比较匹配时 ECCP 引脚变为高电平，请计算 ECCP1CON 寄存器的值。
53. 如果采用半桥，P1A 和 P1C 为高电平有效，其余的是低电平有效，那么请计算 ECCP1CON 寄存器的值。
54. 在 ECCPR1L 和 ECCPR1H 中，哪个寄存器用来设置占空比？
55. 哪个寄存器包含 EDC1B2：EDC1B1？
56. 在设置占空比时，EDC1B2：EDC1B1 的作用是什么？
57. 如果占空比的小数部分是 0.5，那么 EDC1B2：EDC1B1 的值是什么？
58. 在 PWM 编程时，赋给 ECCPRL1 的值通常是 PR2 的_______(分数，倍数)。
59. 假定 PWM 频率为 2 kHz，25% 占空比，XTAL = 10 MHz。请计算 PR2、ECCP1RL 和 EDCB1B2：DC1B1 的值。
60. 假定 PWM 频率为 1.8 kHz，25% 占空比，XTAL = 10 MHz。请计算 PR2、ECCP1RL 和 EDCB1B2：DC1B1 的值。

61. 假定PWM频率为1.5 kHz,25%占空比,XTAL = 10 MHz,请计算PR2、ECCP1RL和EDCB1B2:DC1B1的值。
62. 假定PWM频率为1.2 kHz,25%占空比,XTAL = 10 MHz,请计算PR2、ECCP1RL和EDCB1B2:DC1B1的值。

600

复习题答案

15.1 节

1. 正确。 2. 正确。 3. 正确。 4. RC2(PORTC.2)

15.2 节

1. 错误。 2. 正确。 3. 正确。 4. T3CON

15.3 节

1. 正确。 2. 正确。 3. 错误。 4. 错误。 5. T3CON

15.4 节

1. 正确。 2. 1 3. 正确。 4. 正确。 5. RC2 6. 正确。

15.5 节

1. 错误。最多4个引脚。 2. 1 3. 正确。 4. 错误。 5. RD4 ~ RD7

601

第 16 章 SPI 协议和 DS1306RTC 接口

学习目标：

- ☐SPI（串行外围接口）协议
- ☐SPI 的读写操作原理
- ☐SPI 引脚 SDO、SDI、CE 和 SCLK 的检测
- ☐SPI 的汇编编程和 C 编程
- ☐RTC(实时时钟)芯片的工作原理
- ☐DS1306 RTC 引脚的功能
- ☐DS1306 RTC 寄存器的功能
- ☐DS1306 RTC 与 PIC18 的接口
- ☐显示时间和日期的汇编编程和 C 编程
- ☐RTC 的报警和中断特征的编程

本章讨论 SPI 总线和 DS1306 实时时钟(RTC)(即 SPI 芯片)的接口和编程。在 16.1 节中将介绍 SPI 总线的连接和协议。在 16.2 节中将描述 DS1306 实时时钟(RTC)的引脚功能，并给出它与 PIC18 的接口和编程。DS1306 的 C 语言编程将 16.3 节中讨论。DS1306 的报警特征将在 16.4 节中讨论。

16.1 SPI 总线协议

SPI(串行外围接口)是用于连接设备的一种总线接口，该接口已整合到许多设备中，如 ADC、DAC 和 EEPROM。在这一节中将研究 SPI 总线的引脚和 SPI 的读写操作。

16.1.1 SPI 总线

SPI 总线最初是由 Motorola 公司研发的，但是近年来变成了一个被许多半导体芯片公司所采用的标准。SPI 设备只有两个引脚用于数据传输，分别被称作 SDI(Din)和 SDO(Dout)，而不是像传统总线那样使用 8 个或者更多的引脚。数据引脚的减少极大地降低了封装尺寸和功率消耗，这使得许多 SPI 在存储空间是主要考虑因素的应用中成为理想的选择。SPI 总线由 SCLK(移位时钟)引脚来同步两个芯片间的数据传输。SPI 总线的最后一个引脚是 CE(芯片使能)，是用来开始和结束数据传输的。这四个引脚(SDI、SDO、SCLK 和 CE)使得 SPI 成为一个四线接口。如图 16-1 所示。另外，还有一个被称作三线接口总线的标准。在三线接

口总线中,也包括 SCLK 和 CE,但是只有一个引脚用于数据传输。当 SDI 和 SDO 数据引脚被连接在一起的时候,SDI 四线总线就变成了三线接口。然而,SPI 和三线设备在数据传输协议中仍有许多主要的区别。因此,为了能被用作三线设备,该设备必须在内部支持三线协议。许多像 DS1306 RTC(实时时钟)这样的设备都同时支持 SPI 和三线设备。

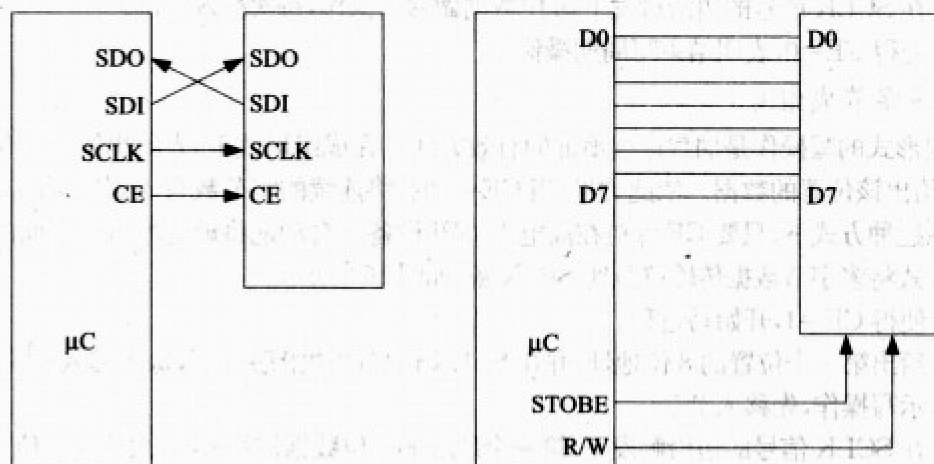


图 16-1 连接到微控制器的 SPI 总线与传统的并行总线

604

16.1.2 SPI 读写协议

在使用 SPI 总线将一个设备与微控制器连接时,微控制器被当作主机,而该 SPI 设备作为从机。因此,要求微控制器产生 SCLK 信号,并将 SCLK 输出到 SPI 设备的引脚。SPI 协议使用 SCLK 信号来同步传输的信息,一次传输一个比特,先传最高有效位(MSB)。在传输期间,CE 必须保持高电平。信息(数据和地址)以 8 比特一组的形式在微控制器和 SPI 设备之间传输,地址字节在前,数据字节在后。为了区别读和写,地址字节中的 D7 位为 1 表示写,而 A7 位为 0 则表示读。这将在下面看到。

16.1.3 将数据写入 SPI 设备的步骤

在访问 SPI 设备时,可以使用两种方式:单字节方式和多字节方式。下面将分别地阐述这两种方式。

1. 单字节写

下面的步骤是以单字节方式将数据发送(写入)到 SPI 设备,如图 16-2 所示。

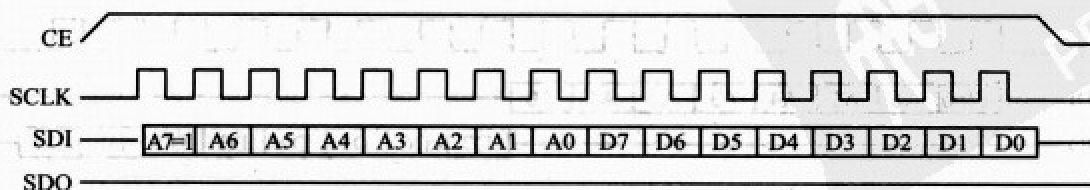


图 16-2 SPI 单字节写操作时序图(注意 A7=1)

(1) 使得 CE=1,开始写过程。

(2) 给出8位地址,并在 SCLK 信号的边沿触发下,每次移入一位。注意,A7=1 表示写操作,先移入 A7。

(3) 在8位地址都传输进去之后,SPI 设备希望立即接收该地址的数据。

(4) 在 SCLK 信号的边沿触发下,8位数据做移入操作,每次移入一位。

(5) 使得 CE=0,表明结束写周期操作。

2. 多字节成组写

成组形式的写操作是加载连续地址的有效方法。在成组形式下,先给出第一个位置的地址,接着给出该位置的数据。依此类推,当 CE=1 时,将连续的字节数写入到连续的存储地址中去。在这种方式下,只要 CE 保持在高电平,SPI 设备会自动地将地址增量。下面的步骤是以成组方式将多字节数据传输(写)到 SPI 设备,如图 16-3 所示。

605

(1) 使得 CE=1,开始写过程。

(2) 给出第一个位置的8位地址,并在 SCLK 信号的边沿触发下,每次移入一位。注意,A7=1 表示写操作,先移入 A7。

(3) 在 SCLK 信号的边沿触发下,第一个地址的8位数据被移入,每次移入一位。依此类推,将连续字节的数据放置到连续的存储空间里。在这个过程中,CE 必须保持高电平,以表明这是一个成组模式的多字节写操作。

(4) 使得 CE=0,表明结束写操作。

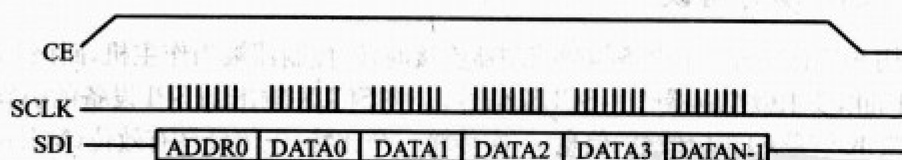


图 16-3　SPI 成组(多字节)模式写操作

16.1.4　从 SPI 设备读数据的步骤

在读取 SPI 设备时,也可以使用两种操作方式:单字节方式和多字节方式。下面将分别阐述这两种方式。

1. 单字节读

下面的步骤是以单字节方式从 SPI 设备中获取(读)数据,如图 16-4 所示。

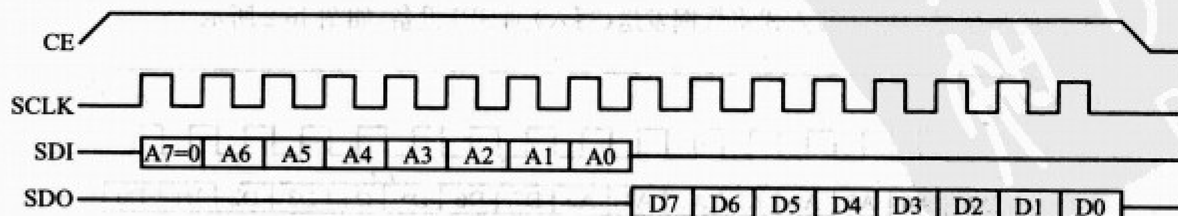


图 16-4　SPI 单字节读操作时序图(注意 A7=0)

(1) 使得 CE=1,开始写过程。

(2) 给出 8 位地址,并在 SCLK 信号的边沿触发下,每次移入一位。注意,A7=0 表示读操作,先移动 A7。

(3) 在 8 位地址都传输进去之后,SPI 设备将送出该地址上的数据。

(4) 在 SCLK 信号的边沿触发下,8 位数据被移出,每次移出一位。

606

(5) 使得 CE=0,结束读周期操作。

2. 多字节成组读

成组形式的读操作是读取连续地址内容的有效方法。在成组形式下,只给出第一个位置的地址。此后,当 CE=1 时,连续的字节将从连续的存储位置被读出。在这种方式下,只要 CE 保持在高电平,SPI 设备内部将自动地增量地址。下面的步骤被用来以成组方式从 SPI 设备获得(读)多字节数据,如图 16-5 所示。

(1) 使得 CE=1,开始写过程。

(2) 给出第一个位置的 8 位地址,并在 SCLK 信号的边沿触发下,每次移入一位。注意,A7=0 表示读操作,先移入 A7。

(3) 在 SCLK 信号的边沿触发下,第一个位置的 8 位数据被移出,每次移出一位。依此类推,就可以很简单地将连续字节的数据从连续的存储空间中读出来。在这个过程中,CE 必须保持高电平,以表明这是一个成组模式的多字节读操作。

(4) 使得 CE=0,表明结束读操作。

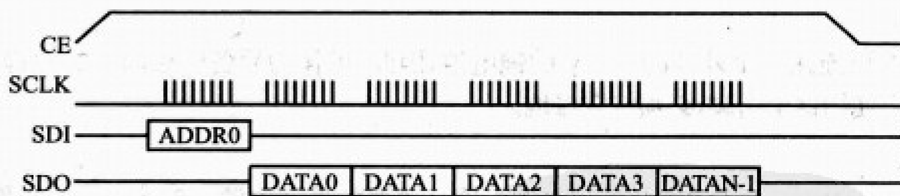


图 16-5 SPI 成组(多字节)模式读操作

16.1.5 复习题

1. 判断正误:SPI 协议是以 8 位一组的形式进行写和读信息的。
2. 判断正误:在 SPI 中,数据信息是紧跟在地址信息之后的。
3. 判断正误:在 SPI 写周期中,地址的 A7 位是 0。
4. 判断正误:在 SPI 写周期中,LSB 先被移入。
5. 请从 CE 信号的角度来说明单字节和成组模式的区别。

607

16.2 DS1306 RTC 接口和编程

实时时钟(RTC)是一个用来为许多应用提供精确的时间和日期的设备,像 X86,IBM PC 的许多系统在主板上都有这个芯片,RTC 芯片提供时、分、秒这样的时间组件,另外还提供年、月、日这样的日期组件。许多 RTC 芯片都有一个内置电池,以保证断电后仍能保持时间和日期。尽管许多像 DSC5000T 这样的微控制器已经内嵌了 RTC,但仍然需要将它们中的大部分连接到外面的 RTC 芯片上。一个最广泛使用的 RTC 芯片是 DS12887,它是 Dalas 半导体公

司生产的。在大多数X86PC机上都安装了这个芯片。最初的IBM PC/AT所用的芯片是Motorola公司的MC14618B RTC。DS12887则是它的替代者,它使用一个内置的锂电池,可在没有外接电源的情况下运作10年。DS12887是一个有8个引脚数据总线的并行RTC。本章将讨论带有SPI总线的DS1306 RTC的接口和编程。根据Maxim公司提供的DS1306数据表可知,它能跟踪"秒、分、时、星期、日、月、年(考虑了闰年,直到2099年都有效)"。DS1306 RTC只提供BCD形式的上述信息。它支持12小时和24小时模式,其中12小时模式提供AM和PM。它不支持白天存储时间的选择,DS1306总共有128 B的固化RAM,它的时钟和控制寄存器占RAM的28 B,而RAM中的剩下的其他96 B可用作通用数据存储。接下来将描述DS1306的引脚,如图16-6所示。

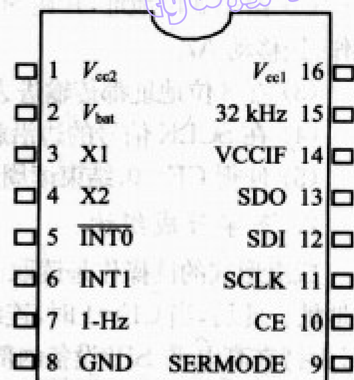


图16-6 DS1306 RTC芯片(来自Maxim/Dallas半导体公司)

1. V_{cc2}

引脚1提供了连到芯片的外接备份电源。该引脚被连接到外部的可充电电源上。这个选择被称作*涓流充电*。如果不使用这个引脚,必须将它接地。

2. V_{bat}

引脚2可连接一个外部的+3 V的锂电池,因此,可作为后备电源向该芯片提供外部电源。如果不使用这个引脚,必须将它接地。

3. V_{cc1}

引脚16被用作芯片的主要外部电源。这个主要的外部电源一般为+5 V。当V_{cc1}降到V_{bat}电压水平之下时,DS1306将切换到V_{bat},由外部的锂电池向RTC提供电源。由DS1306数据表可知,在上电期间,当V_{cc1}比V_{bat}大0.2 V时,该设备将从V_{bat}切换到V_{cc1}。因为在V_{bat}引脚上连接了标准的3 V锂电池,V_{cc1}的电压水平必须保持在3.2 V之上,这是为了使V_{cc1}能够作为芯片的主要电源,这种RTC的固化能力起到了防止数据丢失的作用,如图16-7所示。

608

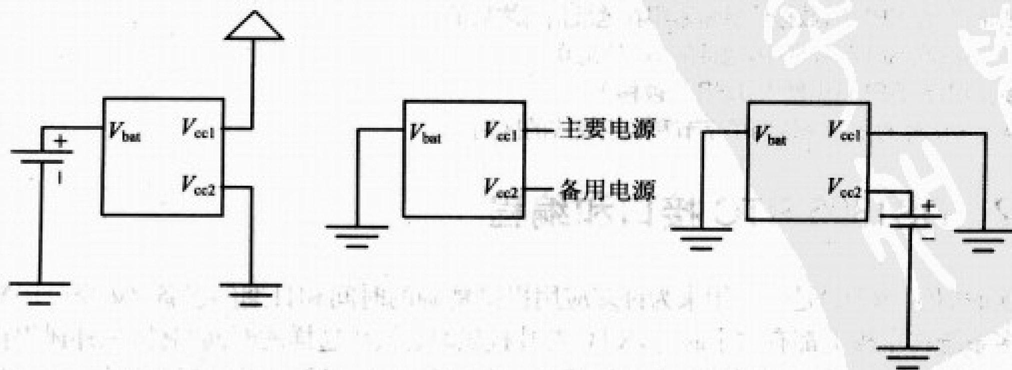


图16-7 DS1306电源连接(来自Maxim/Dallas半导体公司)

4. GND

引脚 8 是地线。

5. SDI(串行数据输入)

SDI 引脚提供了将数据输入到芯片的通道，每次传送一位。

6. SDO(串行数据输出)

SDO 引脚提供了将数据从芯片输出的通道，每次传送一位。

7. 32 kHz

这是一个提供 32.768 kHz 频率的输出引脚，这个频率信号总是出现在引脚上。

8. $X_1 \sim X_2$

它们是输入引脚，允许 DS1306 连接到外部的晶体振荡器上，为芯片提供时钟源，注意，必须使用标准的 32.768 kHz 石英晶体。时钟的精确性取决于该晶体振荡器的质量，如图 16-8 所示。发热将会引起振荡器的漂移。为了避免这种情况，可以使用 DS32 kHz 芯片，它将根据温度的变化自动进行调整。注意，在使用 DS32 kHz 或者相似的时钟发生器时，只需要将它们连接到 X_1 上，因为不需要 X_2 返回信号。

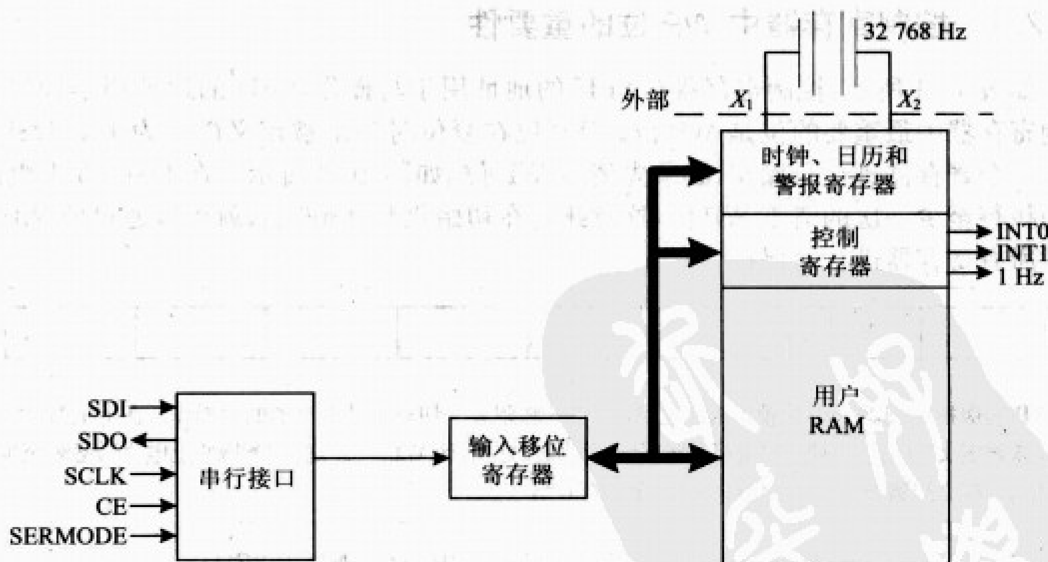


图 16-8 DS1306 的简化框图(来自 Maxim/Dallas 半导体公司)

9. SCLK(串行时钟)

这是一个用作串行时钟的输入引脚，用来同步 DS1306 和微控制器之间的数据传输。

10. 1-Hz

这是一个输出引脚，提供 1 Hz 的方波频率。DS1306 自动地产生 1 Hz 的方波。然而，为了使这 1 Hz 的频率出现在引脚上，必须使能 DS1306 控制寄存器中相关的位。

11. CE

芯片使能(CE)是一个输入引脚，高电平有效。在执行读操作和写操作时 CE 必须保持高

电平。

12. INT0#

该中断请求是一个输出引脚，低电平有效。为了使用INT0，RTC控制器中的中断使能位必须置为高电平。DS1306的中断特征将在16.4节中讨论。

13. INT1

该中断请求是一个输出引脚，高电平有效，为了使用INT1，RTC控制器中的中断使能位609必须置为高电平。DS1306的中断特征将在16.4节中讨论。

14. SERMODE(串行模式选择)

引脚9是一个输入引脚。如果是高电平，就是选择了SPI模式；如果将它接地，就是选择了三线模式。在本书的应用中，SERMODE引脚被连接到了V_{cc}引脚，这是因为这里使用SPI协议来对1306芯片进行编程。

15. V_{ccif}

引脚14是一个供电输入的逻辑接口。这个引脚允许DS1306与混合系统中的3V逻辑系统进行连接。如果在系统中使用5V以外的电源，可以参考DS1306数据表。

16.2.1　控制寄存器中WP位的重要性

如表16-1所示，控制寄存器有8FH的地址用于写操作，0FH的地址用于读操作。控制寄存器中最重要的位是WP位。WP位在复位时是未被定义的。为了对DS1306的任一个寄存器进行写操作，必须先清零WP位，如图16-9所示。在DS1306上电时，必须执行至少一次的清零WP位，这意味着在初始化DS1306后，就可以通过使WP=1来对所有寄存器进行写保护。

	WP						

WP(写保护)　如果WP位被置为高电平，DS1306将阻止一切对于其寄存器的写操作。在上电时，WP位未被定义。因此，必须在对寄存器实行任何写操作之前，让WP=0。这在DS1306上电之后必须至少执行一次。控制寄存器的其他位将会在下一节介绍

610

图16-9　DS1306控制寄存器的WP位(写地址为8FH)

16.2.2　DS1306的地址映射

DS1306总共有128 B的RAM空间，地址是00～7FH。开头的15个地址区00～0E专门用于时间、日期和警报数据的RTC值。接下来的3 B用作控制寄存器和状态寄存器，它们位于0F～11地址。接下来的14 B(12H～1FH)被保留(未被使用)。还剩下96 B(也就是从地址20H～7FH)被用作通用目的数据存储。因此，RAM的整个128 B除了从12～1FH的地址外，其余的均可被直接读或者写。表16-1给出了DS1306的地址映射。在本节中将学习时间和日期，而警报将在16.4节中介绍。

表 16-1 DS1306 的寄存器(摘自数据表)

十六进制地址		D7	D6	D5	D4	D3	D3	D1	D0	范围
READ	WRITE									十六进制
0x00	0x80	0	10 秒			秒				00~59
0x01	0x81	0	10 分			分				00~59
0x02	0x82	0	24/12	20 小时 P/A	10HR	小时				00~23 01~12P/A
0x03	0x83	0	0	0	0	0	天			01~07
0x04	0x84	0	0	10 日期		日期				01~31
0x05	0x85	0	0	10 月		月				01~12
0x06	0x86	0	10 年			年				00~99
0x07	0x87	M	10 秒报警 0			秒报警 0				00~59
0x08	0x88	M	10 分报警 0			分报警 0				00~59
0x09	0x89	M	24/12	20 小时 P/A	10 小时	小时报警 0				00~23 01~12 P/A
0x0A	0x08A	M	0	0	0	0	天报警 0			01~07
0x0B	0x8B	M	10 秒报警 1			秒报警 1				00~59
0x0C	0x8C	M	10 分报警 1			分报警 1				00~59
0x0D	0x8D	M	24/12	20 小时 P/A	10 小时	小时报警 1				00~23 01~12 P/A
0x0E	0x8E	M	0	0	0	0	天报警 1			01~07
0x0F	0x8F	控制寄存器								
0x10	0x90	状态寄存器								
0x11	0x91	涓流充电器寄存器								
0x12-0x1F	0x92-0x9F	保留								

16.2.3 时间和日期地址的位置和模式

字节地址 0~6 被用作时间和日期,如表 16-2 所示。表 16-2 是从表 16-1 中提取出来的。它简要地给出了读写模式下的地址位置以及每个地址的数据范围。DS1306 只提供了 BCD 码形式的数据。注意用于小时模式的数据范围。可以通过设置小时位置 02 的第 6 位来选择是 12 小时模式还是 24 小时模式。当 D6=1 时,选择的是 12 小时模式。当 D6=0 的时候,选择的是 24 小时模式。在 12 小时模式下,通过第 5 位来决定是 AM 和还是 PM。如果 D5=0,就是选择了 AM。如果 D5=1,就是选择了 PM。例 16-1 说明了如何获取小时位置可接收的数据范围。

611

例 16-1 利用表 16-1,确定表 16-2 中的小时位置的值。

解:(a) 对于 24 小时模式,有 D6=0。因此,范围是 0000 0000~0010 0011,它们的 BCD 码形式就是00~23。

(b) 对于 12 小时模式，让 D6=1、D5=0，就代表是 AM。因此，范围是 010 **0 0001**~010**1 0010**，它们的 BCD 码形式就是 41~52。

(c) 对于 12 小时模式，让 D6=1、D5=1，就代表是 PM。因此，范围是 011**0 0001**~011**1 0010**，它们的 BCD 码形式就是 61~72。

表 16-2　DS1306 用于时间和日期的地址位置(摘自表 16-1)

十六位地址 读	位置 写	功　能	日期范围	十六进制范围
00	80	秒	00~59	00~59
01	81	分钟	00~59	00~59
02	82	小时，12 小时模式	01~12	41~52AM
		小时，12 小时模式	01~12	61~72PM
		小时，24 小时模式	00~23	00~23
03	83	星期中的天，周日=1	01~07	01~07
04	84	月中的天	01~31	01~31
05	85	月	01~12	01~12
06	86	年	00~99	00~99

16.2.4　使用 MSSP 模块来连接 PIC18 和 DS1306

DS1306 同时支持 SPI 和三线模式。在 DS1306 中，通过连接 SERMODE 引脚到 V_{cc} 来选择 SPI 模式。如果 SERMODE=GND，使用三线协议。本节只使用 SPI 模式。PIC18 内部的 MSSP(主机同步串行端口)模块支持 SPI 总线协议。MSSP 模块有 3 个与 SPI 相关的寄存器，它们分别是 SSPBUF、SSPCON1 和 SSPSTAT。为了传输一个字节的数据，先将其放在 SSPBUF 里。SSPBUF 寄存器将保存通过 SPI 总线收到的字节。图 16-10 和图 16-11 介绍了 SPI 接口的其他两个寄存器。可以使用 SSPCON1 来选择 SPI 模式。注意 SSPCON1 寄存器中的 SSPEN 位必须被置为高电平，才能允许 PIC18 的引脚用于 SPI 数据总线协议。还必须使用 SSPCON1 中的 SSPM3：SSPM0 位来选择 SPI 主机模式。在本书的应用中，使用 $F_{osc}/64$ 的速度可以在 PIC18 和 DS1306 RTC 之间的数据传输中达到最佳性能。

在设置了 SSPCON1 之后，必须选择合适的位用于 SSPSTAT 寄存器的时序，如图 16-11 所示。在本书的应用中，在上升沿将数据传给 SPI 设备，在 SCLK 时钟脉冲的中间将数据从 SPI 设备中读入。

因为使用了 PIC18 的 SPI 特征来与 SPI 设备通信，所以必须使用 SPI 信号的指定引脚，它们是 RC2(CE)、RC3(SCLK)、RC4(SDI)和 RC5(SDO)，如图 16-12 所示。

		SSPEN		SSPM3	SSPM2	SSPM1	SSPM0

SSPEN D5 同步串行端口使能位
1=使能串行端口并设置 SCK、SDO、SDI 作为串行端口引脚
0=关闭串行端口引脚并设置这些引脚为 I/O 端口

SSPM3：SSPM0 D3～D0 SPI 模式选择位
0010= SPI 主机，时钟$=F_{osc}/64$
0001= SPI 主机，时钟$=F_{osc}/16$
0000=SPI 主机，时钟$=F_{osc}/4$

其余的位在 SPI 的实现中并未用到
这里在主模式下使用 SPI
注意：显示的部分只用于 SPI。

图 16-10 SSPCON1-SSP 控制寄存器 1

SMP	CKE						BF

SMP D7 采样位
1=在数据输出结束后对输入数据采样
0=在数据输出还在进行时对输入数据采样

CKE D6 SPI 时钟边沿选择位
1=在激活到空闲时钟状态的过渡阶段发送数据
0=在空闲到激活时钟状态的过渡阶段发送数据

BF D0 缓冲区满状态位，只用作接收
1=接收完毕，SSPBUF 已满
0=接收未完成，SSPBUF 是空的

其余的各位被用作 I²C 模块
注意：显示的部分只用于 SPI。

图 16-11 SSPSTAT-SSP 状态寄存器 1

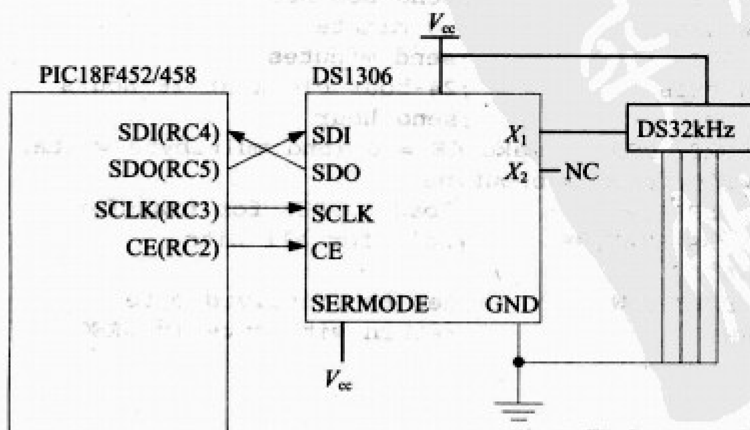


图 16-12 DS1306 与 PIC18 的连接

16.2.5 使用汇编设置时间

程序16-1使用24小时时钟模式，初始化时钟为16：58：55。它使用单字节模式来写DS1306的控制寄存器，使用多字节成组模式来写秒、分和时。关于程序16-1的SPI子例程，必须要注意以下几点。

(1) 为了PIC18能够使用SPI协议传输字节数据，必须将数据放在SSPBUF里。

(2) 在写SSPBUF后，必须监视SSPSTAT寄存器的BF标志位来确认整字节已被传输。

(3) SSPBUF也可用作从SPI设备接收到的数据的目的寄存器。这将发生在数据被发送时。BF标志位用来指示整个字节已被接收。

程序 16-1

```
;Program 16-1: Setting the Time
      MOVLW  0x00
      MOVWF  SSPSTAT ;read at middle, send on active edge
      MOVLW  0x22
      MOVWF  SSPCON1 ;enable master SPI, Fosc / 64
      CLRF  TRISC           ;make PORTC output
      BSF  TRISC,SDI        ;except SDI
 ;--  send control byte to DS1306 in single-byte mode
      BSF   PORTC,RC2       ;make CE = 1 for single-byte
      CALL  SDELAY
      MOVLW  0x8F           ;DS1306 control register address
      CALL  SPI
      MOVLW  0x00           ;clear WP bit for write
      CALL  SPI
      BCF  PORTC,RC2
;make CE = 0 to end write (single-byte)
      CALL  SDELAY
;--   send the data to DS1306 in burst mode
      BSF   PORTC,RC2  ;make CE = 1 (start multibyte write)
      MOVLW  0x80           ;seconds register address
      CALL  SPI             ;send address
      MOVLW  0x55           ;55 seconds
      CALL  SPI             ;send seconds
      MOVLW  0x58           ;58 minutes
      CALL  SPI             ;send minutes
      MOVLW  0x16           ;24-hour clock at 16 hours
      CALL  SPI             ;send hour
      BCF  PORTC,RC2   ;make CE = 0 (end multibyte write)
;--   SPI write/read subroutine
SPI   MOVWF  SSPBUF         ;load SSPBUF for transfer
WAIT  BTFSS  SSPSTAT,BF     ;wait for all bits
      BRA  WAIT
      MOVF  SSPBUF,W        ;get the received byte
      RETURN                ;return with byte in WREG
      END
```

614

16.2.6 使用汇编设置日期

程序16-2说明了如何设置日期为2004年10月19日。

程序 16-2

```
;Program 16-2: Setting the Date
        MOVLW  0x00
        MOVWF  SSPSTAT ;read at middle, send on active edge
        MOVLW  0x22
        MOVWF  SSPCON1          ;master SPI enable, Fosc / 64
        CLRF  TRISC             ;make PORTC output
        BSF  TRISC,SDI          ;except SDI
        BSF   PORTC,RC2         ;enable the RTC
        MOVLW  0x8F             ;DS1306 control register address
        CALL  SPI
        MOVLW  0x00             ;clear WP bit for write
        CALL  SPI
        BCF  PORTC,RC2          ;turn off RTC
;-- send the date to DS1306
        BSF   PORTC,RC2         ;enable the RTC
        MOVLW  0x84             ;date register address
        CALL  SPI               ;send address
        MOVLW  0x19             ;19th of the month
        CALL  SPI               ;send date
        MOVLW  0x10             ;October
        CALL  SPI               ;send month
        MOVLW  0x04             ;2004
        CALL  SPI               ;send year
        BCF  PORTC,RC2          ;disable RTC
 ;-- SPI write/read subroutine
SPI  MOVWF  SSPBUF              ;load SSPBUF for transfer
WAIT BTFSS  SSPSTAT,BF          ;wait for all bits
     BRA  WAIT
     MOVF  SSPBUF,W             ;get the received byte
     RETURN                     ;return with byte in WREG
     END
```


16.2.7 RTC 设置、读取和显示时间和日期

程序 16-3 是设置、读取和显示时间和日期的完整汇编程序代码。在将时间和日期从压缩 BCD 转换为 ASCII 码后，通过串行端口将它们显示在 IBM PC 的屏幕上。

程序 16-3

```
;Program 16-3
#include p18f458.inc
D1uL  EQU  D'2'                 ;1 microsecond delay byte
DR1uL EQU  0x0D         ;register for 1 microsecond delay
DAY   EQU  10H                  ;for day of the week
MON   EQU  11H                  ;fileReg starting with month
DAT   EQU  12H                  ;for day of the month
YR    EQU  13H                  ;for year
HR    EQU  14H                  ;for hour
MIN   EQU  15H                  ;for minutes
SEC   EQU  16H                  ;for seconds
CNT   EQU  20H                  ;for counter
```

```
TMP   EQU 21H                ;for conversions
      MOVLW 0x00
      MOVWF SSPSTAT ;read at middle, send on active edge
      MOVLW 0x22
      MOVWF SSPCON1          ;master SPI enable, Fosc / 64
      CLRF TRISC             ;make PORTC output
      BSF TRISC,SDI          ;except SDI
      BSF TRISC,RX           ;and RX
;-- enable USART communication
      MOVLW B'00100000'      ;enable transmit and low baud
      MOVWF TXSTA            ;write to reg
      MOVLW D'15'        ;9600 bps (Fosc / (64 * Speed) - 1)
      MOVWF SPBRG            ;write to reg
      BCF TRISC, TX      ;make TX pin of PORTC an output pin
      BSF RCSTA, SPEN        ;enable the serial port
;-- start a new line for USART communications
      MOVLW 0x0A             ;form feed
      CALL TRANS
      MOVLW 0x0D             ;new line
      CALL TRANS
;-- send control byte to DS1306
      BSF   PORTC,RC2        ;enable the RTC
      CALL SDELAY
      MOVLW 0x8F             ;control register address
      CALL SPI
      MOVLW 0x00             ;clear WP bit for write
      CALL SPI
      BCF PORTC,RC2          ;disable RTC
      CALL SDELAY
;-- send the time followed by date
      BSF   PORTC,RC2        ;enable the RTC
      MOVLW 0x80         ;seconds register address for write
      CALL SPI               ;send address
      MOVLW 0x55             ;55 seconds
      CALL SPI               ;send seconds
      MOVLW 0x58             ;58 minutes
      CALL SPI               ;send minutes
      MOVLW 0x16             ;24-hour clock at 16 hours
      CALL SPI               ;send hour
      MOVLW 0x3              ;Tuesday
      CALL SPI               ;send day of the week
      MOVLW 0x19             ;19th of the month
      CALL SPI               ;send day of the month
      MOVLW 0x10             ;October
      CALL SPI               ;send month
      MOVLW 0x04             ;2004
      CALL SPI               ;send year
      BCF PORTC,RC2          ;disable RTC
      CALL SDELAY
;-- get the time and date from DS1306
RDA   BSF   PORTC,RC2        ;enable the RTC
```

616

```
        CALL SDELAY
        MOVLW 0x00          ;seconds register address for read
        CALL SPI                ;send address to DS1306
        CALL SPI                ;start getting time/date
        MOVWF SEC               ;save the seconds
        CALL SPI                ;get the minutes
        MOVWF MIN               ;save the minutes
        CALL SPI                ;get the hour
        MOVWF HR                ;save the hour
        CALL SPI                ;get the day
        MOVWF DAY               ;save the day
        CALL SPI                ;get the date
        MOVWF DAT               ;save the date
        CALL SPI                ;get the month
        MOVWF MON               ;save the month
        CALL SPI                ;get the year
        MOVWF YR                ;save the year
        BCF PORTC,RC2           ;disable RTC
;-- convert packed BCD to ASCII and display
        LFSR FSR0,0x11      ;address of fileReg for time/date
        MOVLW D'6'              ;6 bytes of data to display
        MOVWF CNT               ;set up the counter
SND     MOVFF INDF0,TMP         ;get the data for high nibble
        MOVLW 0xF0              ;clear low nibble
        ANDWF TMP,F             ;keep in TMP register
        SWAPF TMP,F             ;switch high and low nibbles
        MOVLW 0x30              ;convert to ASCII
        IORWF TMP,W             ;put in WREG
        CALL TRANS              ;display the data
        MOVFF  POSTINC0,TMP     ;get the data and point to next
        MOVLW 0x0F              ;clear high nibble
        ANDWF TMP,F             ;keep in TMP register
        MOVLW 0x30              ;convert to ASCII
        IORWF TMP,W             ;put in WREG
        CALL TRANS              ;display the data
        MOVLW ':'
        CALL TRANS
        DECFSZ CNT              ;Is it the last one?
        BRA SND                 ;no
        MOVLW 0x0D              ;line feed
        CALL TRANS
        BRA RDA         ;keep reading time/date and display them
;-- SPI write/read subroutine
SPI     MOVWF SSPBUF            ;load SSPBUF for transfer
WAIT    BTFSS SSPSTAT,BF        ;wait for all bits
        BRA WAIT
        MOVF SSPBUF,W           ;get the received byte
        RETURN                  ;return with byte in WREG
;----serial data transfer subroutine
TRANS   BTFSS PIR1, TXIF        ;wait until the last bit is gone
        BRA TRANS               ;stay in loop
        MOVWF TXREG         ;load the value to be transmitted
        RETURN                  ;return to caller
```

617

```
;----short delay
SDELAY: MOVLW   D1uL            ;low byte of delay
        MOVWF DR1uL             ;store in register
DS1     DECF DR1uL,F            ;stay until DR1uL becomes 0
        BNZ   DS1
        RETURN
        END
```

16.2.8 复习题

1. 判断对错:DS1306 的 RAM 的所有内容都是非易失性的。
2. DS1306 的 RAM 有多少字节被用作时钟和日期?
3. DS1306 的 RAM 有多少字节被用作通用目的应用?
4. 判断对错:DS1306 的 D_{in}是单引脚的。
5. DS1306 的哪一个引脚被用作 SPI 连接中的时钟?
6. 判断对错:为了使 DS1306 运行在 SPI 模式,必须使 SERMODE=GND。

618

16.3 DS1306 RTC 的 C 编程

本节将使用 PIC18 C 语言对 DS1306 编程。在学习本节之前,请回顾 16.1 节介绍过的 DS1306 芯片的基础知识。

16.3.1 使用 C 语言设置时间和日期

程序 16-4C 将说明如何在 DS1306 配置中设定时间和日期,如图 16-2 所示。

程序 16-4C

```
//Program 16-4C : Setting time and date
#include <p18f458.h>
unsigned char SPI(unsigned char);
void SDELAY(int ms);
void main()
  {
    SSPSTAT = 0; //read at middle, send on active edge
    SSPCON1 = 0x22;        //master SPI enable, Fosc / 64
    TRISC = 0;             //make PORTC output
    TRISCbits.TRISC4 = 1; //except SDI
    TRISCbits.TRISC7 = 1; //and RX
    PORTCbits.RC2 = 1;     //enable the RTC
    SDELAY(1);
    SPI(0x8F);             //control register address
    SPI(0x00);             //clear WP bit for write
    PORTCbits.RC2 = 0;     //end of single-byte write
    SDELAY(1);
    PORTCbits.RC2 = 1;     //begin multibyte write
    SPI(0x80);             //seconds register address
    SPI(0x55);             //55 seconds
    SPI(0x58);             //58 minutes
```

```
        SPI(0x16);              //24-hour clock at 16 hours
        SPI(0x3);               //Tuesday
        SPI(0x19);              //19th of the month
        SPI(0x10);              //October
        SPI(0x04);              //2004
        PORTCbits.RC2 = 0;      //end multibyte write
        SDELAY(1);
        }
//-- SPI Write/Read subroutine
unsigned char SPI(unsigned char myByte)
   {
        SSPBUF = myByte;            //load SSPBUF for transfer
        while(!SSPSTATbits.BF);     //wait for all bits
        return SSPBUF;          //return with received byte
        }
```

619

16.3.2 使用 C 语言读取和显示时间和日期

程序 16-5C 将说明如何读取时间,并转换成 ASCII 码,然后通过串行端口将 ASCII 数据发送到 PC 屏幕上。

程序 16-5C

```
//Program 16-5C : Reading and Displaying Time
#include <p18f458.h>
unsigned char SPI(unsigned char);
void TRANS(unsigned char);
void BCDtoASCIIandSEND(unsigned char);
void SDELAY(int ms);

void main()
  {
   unsigned char data[7];   //holds date and time
   unsigned char tmp;       //for BCD to ASCII conversion
   int i;
   SSPSTAT = 0;         //read at middle, send on active edge
   SSPCON1 = 0x22;          //master SPI enable, Fosc / 64
   TRISC    = 0;            //make PORTC output
   TRISCbits.TRISC4 = 1;    //except SDI
   TRISCbits.TRISC7 = 1;    //and RX
   TXSTA = 0x20;            //enable transmit and low baud
   SPBRG = 15;          //9600 bps (Fosc / (64 * Speed) - 1)
   RCSTAbits.SPEN = 1;      //enable the serial port
   TRANS(0x0A);             //form feed
   TRANS(0x0D);             //new line
//-- get the time and date from RTC and save them
  while(1)
    {
     PORTCbits.RC2 = 1;     //begin multibyte read
     SDELAY(1);
     SPI(0x00);             //seconds register address
```

```
        for(i=0;i<7;i++)
          {
          data[i] = SPI(0x00); //get time/date and save
          }
        PORTCbits.RC2 = 0;      //end of multibyte read
    //-- convert time/date and display MM:DD:YY:HH:MM:SS
        BCDtoASCIIandSEND(data[5]); //the month
        BCDtoASCIIandSEND(data[4]); //the date
        BCDtoASCIIandSEND(data[6]); //the year
        BCDtoASCIIandSEND(data[2]); //the hour
        BCDtoASCIIandSEND(data[1]); //the minute
        BCDtoASCIIandSEND(data[0]); //the second
        TRANS(0x0D);            //new line
        }
      }
    //-- SPI Write/Read
    unsigned char SPI(unsigned char myByte)
      {
      SSPBUF = myByte;
      while(!SSPSTATbits.BF);
      return SSPBUF;
      }
    void TRANS(unsigned char myChar) //serial data transfer
      {
      while(!PIR1bits.TXIF);
      TXREG = myChar;     //load the value to be transmitted
      }
    void BCDtoASCIIandSEND(unsigned char myValue)
      {
      unsigned char tmp = myValue;
      tmp = tmp & 0xF0;         //mask lower nibble
      tmp = tmp >> 4;           //swap it
      tmp = tmp | 0x30;         //make it ASCII
      TRANS(tmp);               //display
      tmp = myValue;            //for other digit
      tmp = tmp & 0x0F;         //mask upper nibble
      tmp = tmp | 0x30;         //make it ASCII
      TRANS(tmp);               //display
      TRANS(':');               //display separator
      }

    void SDELAY(int ms)
      {
      unsigned int i, j;
      for(i=0;i<ms;i++)
        for(j=0;j<135;j++);
      }
```

620

16.3.3　复习题

1. 判断对错:DS1306 的 RAM 的所有内容都是易失性的。
2. DS1306 的 RAM 的哪个位置被用作时钟和日期?
3. DS1306 的 RAM 的哪个位置被用作通用目的应用?

4. 判断对错：DS1306 的 D_{out}是一个单引脚。
5. 判断对错：CE 是一个输出引脚。
6. 判断对错：为了让 DS1306 工作在 SPI 模式，必须让 SERMODE=VCC。

621

16.4 DS1306 的警报和中断特征

在本节中，将使用汇编语言和 C 语言来编程 DS1306 芯片的警报和中断特征。DS1306 的这些强大的特征在许多实际应用中非常有用。在 DS1306 中有两个警报，分别是警报 0 和警报 1，每个警报都有它们自己的硬件中断。另外，还有一个 1 Hz 方波的引脚，接下来将讨论这个引脚。这些特征可以通过如图 16-13 所示的控制寄存器来访问。

1. 对 1-Hz 特征编程

DS1306 的 1-Hz 引脚可以提供一个 1Hz 频率的方波输出。DS1306 在内部会自动产生 1 Hz方波，但是它通常是被阻塞的。因此，必须使能控制寄存器中的 1 Hz 位来让这 1 Hz 方波出现在该引脚上。下面的代码显示了这一过程。因为是向一个单独的地址写操作，所以并未用到成组模式。

0	WP	0	0	0	1-Hz	AIE1	AIE0

WP(写保护) 如果 WP 位被置为高电平，DS1306 将阻止一切对于其寄存器的写操作。在对寄存器实行任何写操作之前，必须让 WP=0。在上电时，WP 位是未被定义的。因此，必须在对寄存器实行任何写操作之前，让 WP=0

1-Hz (1-Hz 输出使能) 如果这个位被置为高电平，它允许从 DS1306 的 1-Hz 引脚输出 1-Hz 的方波。如果将其置为低电平，在 1-Hz 引脚将得到高阻态。注意 1-Hz 的频率是由 DS1306 自动产生的，但是在将这个位设置为高电平之前，它不会出现在 1-Hz 引脚上

AIE0 警报中断 0 使能。如果 AIE0=1，当实时时间(hh:mm:ss)的所有三个字节和警报的 hh:mm:ss 字节相同时，INT0 引脚将被置为低电平。同样，如果 AIE=1，即每秒一次、每分一次、每时一次的情况，将会把 INT0 置为低电平

AIE1 警报中断 1 使能。如果 AIE1=1，当实际时间的所有三个字节和警报的字节相同时，INT0 引脚将被置为高电平。同样，如果 AIE0=1，即每秒一次、每分一次、每时一次的情况，将会把 INT0 置为高电平

图 16-13 DS1306 控制寄存器(写地址为 8FH)

```
    MOVLW 0x00
    MOVWF SSPSTAT     ;middle read, active edge send
    MOVLW 0x22
    MOVWF SSPCON1     ;master SPI enable, Fosc / 64
    CLRF  TRISC       ;make PORTC output
    BSF   TRISC,SDI   ;except SDI
;-- send control byte to enable write first (Figure 16-13)
    BSF   PORTC,RC2   ;enable the RTC
```

622

```
        CALL  SDELAY
        MOVLW  0x8F           ;Control register address
        CALL  SPI
        MOVLW  0x0            ;clear WP bit for write
        CALL  SPI
        BCF    PORTC,RC2      ;disable RTC
        CALL  SDELAY
;-- send control byte to enable 1 Hz signal after WP = 0
        BSF    PORTC,RC2      ;enable the RTC
        CALL  SDELAY
        MOVLW  0x8F           ;Control register address
        CALL  SPI
        MOVLW  0x04  ;enable 1 Hz signal in Control register
        CALL  SPI
        BCF    PORTC,RC2      ;disable RTC
        CALL  SDELAY
```

2. 警报 0、警报 1 和中断

DS1306 芯片中有两个用于一天时间的警报。它们分别是警报 0 和警报 1。可以通过向地址 87H～8AH 的寄存器写入数据来访问警报 0，如表 16-3 所示。对于警报 1，可以通过向地址为 8BH～8EH 的寄存器写入数据来访问，如表 16-3 所示。在每个时钟进行更新时，RTC 将比较时钟寄存器和警报寄存器。当储存在时钟寄存器 0、1 或者 2 中的值与警报寄存器中的值相匹配时，状态寄存器内相应的警报标志位(IRQF0 或 IRQF1)将会变为高电平，如图 16-14 所示。因为查询 IRQXF 太耗时间，所以可以使能控制寄存器中的 AIEX 位，以获得来自 INT0 和 INT1 引脚的硬件中断。

表 16-3 DS1306 用于时间、日历和警报的地址

十六进制地址 读	写	功 能	D7	数据范围 BCD	可能的十六进制范围
00H	80H	秒	0	00～59	00～59
01H	81H	分	0	00～59	00～59
02H	82H	小时，12 小时模式	0	01～12	41～52AM
		小时，12 小时模式	0	01～12	61～72PM
		小时，24 小时模式	0	00～23	00～23
03H	83H	一周的天，周日＝1	0	01～07	01～07
04H	84H	一月的天	0	01～31	01～31
05H	85H	月	0	01～12	01～12
06H	86H	年	0	00～99	00～99
07H	87H	秒警报器 0	0 或 1	00～59	00～59 或 89～A9
08H	88H	分警报器 0	0 或 1	00～59	00～59 或 89～A9
09H	89H	小时警报器 0，12 小时	0 或 1	01～12	41～52 或 C1～A2AM
		小时警报器 0，12 小时	0 或 1	01～12	61～72 或 D1～F2PM
		小时警报器 0，24 小时	0 或 1	00～23	00～23 或 80～A3
0A	8A	天警报器 0	0 或 1	1～7	01～07
0BH	8BH	秒警报器 1	0 或 1	00～59	00～59 或 89～A9
0CH	8CH	分警报器 1	0 或 1	00～59	00～59 或 89～A9
0DH	8DH	小时警报器 1，12 小时	0 或 1	01～12	41～52 或 C1～A2 AM

（续）

十六进制地址 读	写	功能	D7	数据范围 BCD	可能的十六进制范围
		小时警报器1,12小时	0或1	01～12	61～72或D1～F2 PM
		小时警报器1,24小时	0或1	00～23	00～23或80～A3
0EH	8EH	天警报器1	0或1	1～7	01～07
0FH	8FH	控制寄存器			
10H	90H	状态寄存器			
11H	91H	涓流寄存器			
12～1FH	82～9FH	保留			
20～7FH	A0～FFH	96自己的用户RAM			

0	0	0	0	0	0	IRQF1	IRQF0

IRQF0（中断0请求标志位） 当实时时间(hh:mm:ss)的所有三个字节和警报0的hh:mm:ss字节相同时，IRQF0将会被置为高电平，同样，对于每秒一次、每分一次、每时一次的情况，也将会把INT0置为高电平。可以使用查询法来观察IRQF0的状态。然而，在控制寄存器中，如果将AIE0=1，IRQF0会把INT0引脚置为低电平，让它成为一个硬件中断。任何对于警报0寄存器的读或者写操作都将清零IRQF0位

IRQF1（中断1请求标志位） 当实时时间(hh:mm:ss)的所有三个字节和警报1的hh:mm:ss字节相同时，IRQF1将会被置为高电平，同样地，对于每秒一次、每分一次、每时一次的情况，也将会把INT1置为高电平。可以使用查询法来观察IRQF1的状态。然而，在控制寄存器中，如果将AIE1=1，IRQF1会把INT1引脚置为高电平，让它成为一个硬件中断。任何对于警报1寄存器的读或者写操作都将清零IRQF1位

图16-14 状态寄存器(读地址为10H)

3. 警报和IRQ输出引脚

INT0和INT1的警报中断可以被编程为下列的频率出现：(a)每星期一次，(b)每天一次，(c)每小时一次，(d)每分钟一次，(e)每秒一次。接下来将分别讨论这几种情况。

4. 每天一次的警报

表16-3显示了属于警报秒、警报分、警报时、警报天的地址。请注意这些地址的D7位。当天警报器的D7位被置为高电平时，警报将每天产生一次。因此当为每天一次的警报编程时，必须将警报需要的时间写入时、分和秒的警报地址，并将天警报器的D7位置为高电平，如表16-4所示。随着时钟跟随时间，并且当实时时钟的时、分、秒的三个字节与警报的时、分和秒的值相匹配时，DS1306状态寄存器中的IRQXF标志位将变为高电平。可以使用查询法检查状态寄存器中的IRQXF标志位，但这将是对微控制器资源的浪费。或者，在实时时间与警报时间相匹配时，可以允许硬件的INTX引脚被激活。必须注意的是，为了使得DS1306的硬件INTX引脚可用于警报，控制寄存器(AIEX)中用于警报的中断使能位必须被置为高电平。下面将简单分析这个过程。

623～624

5. 每小时一次的警报

为了对每小时响一次的警报编程，必须将天警报寄存器和时警报寄存器的 D7 位置为高电平，如表 16-4 所示。

6. 每分钟一次的警报

为了对每分钟响一次的警报编程，必须将天警报寄存器、时警报寄存器、分警报寄存器的 D7 位置为高电平，如表 16-4 所示。

7. 每秒一次的警报

为了对每秒响一次的警报编程，必须将天警报寄存器、时警报寄存器、分警报寄存器和秒警报寄存器的 D7 位置为高电平，如表 16-4 所示。

8. 一星期一次的警报

为了对每星期响一次的警报编程，必须将天警报寄存器、时警报寄存器、分警报寄存器和秒警报寄存器的 D7 位清零，如表 16-4 所示。

表 16-4 DS1306 的每日时间警报的掩码位

警报寄存器屏蔽位(D7)				
秒	分	时	天	功能
1	1	1	1	每秒警报一次
0	1	1	1	当秒匹配时报警(每分报警一次)
0	0	1	1	当分和秒都匹配时报警(每小时报警一次)
0	0	0	1	当小时、分和秒都匹配时报警(每天报警一次)
0	0	0	0	当小时、分和秒都匹配时报警(每星期报警一次)

例 16-2 使用表 16-4，如果想在 16：05：07 启动警报，并且在以后每分钟报警一次(即在分钟后的第 7 秒报警)，请计算应写入警报 1 寄存器中的值。

解：因为使用的是 24 小时模式，所以有 HR 寄存器中 D6=0。因此，对于 1001 0110，有 BCD 码的形式为 16。这意味着，必须在 DS1306 的寄存器区域 8D 写入 96H。注意，根据表 16-4 可知，D7 的值是 1。

对于 MIN 寄存器，05 的 BCD 码形式是 1000 0101，这意味着必须在 DS1306 的寄存器区域 8C 写入 85H 。注意，根据表 16-4 可知，D7 的值是 1。

对于 SEC 寄存器，07 的 BCD 码形式是 0000 0111，这意味着必须在 DS1306 的寄存器区域 8B 写入 07H 。注意，根据表 16-4 可知，D7 的值是 0。

625

对于每分钟响一次警报，必须保证警报 1 的 D7 位也被置为 1。如表 16-4 所示。

9. 用 DS1306 的 INT0 来触发 PIC18 的中断

将 DS1306 的 INT0 位与 PIC18 的外部中断引脚连接起来，如图 16-15 所示。这样，执行任务可以是每天一次地、每分钟一次地，等等。例 16-2 显示了警报 0 寄存器所需要的值。程序 16-6 使用警报 0 的中断(INT0)向串行端口每分钟发送一次信息 YES，时间为每分钟后的第 8 秒。

程序 16-6

```
;Program 16-6
DluL  EQU   D'2'              ;1 microsecond delay byte
DR1uL EQU   0x0D         ;register for 1 microsecond delay
      ORG   0x00
      BRA   MAIN              ;bypass INT vector table
      ORG   0x08
      BTFSC INTCON,INT0IF     ;Was it INT0?
      BRA   INT0_ISR          ;yes, go to INT0 ISR
      RETFIE
      ORG   0x28
;-- initialize SPI, INT0, and USART
MAIN  CLRF  TRISC             ;make PORTC output
      BSF   TRISC,SDI         ;except SDI
      BSF   TRISC,RX          ;and RX
      BSF   TRISB,INT0        ;make RB0 input for interrupt
      MOVLW 0x00
      MOVWF SSPSTAT           ;middle read, active edge send
      MOVLW 0x22
      MOVWF SSPCON1           ;master SPI enable, Fosc / 64
      BCF   INTCON2,INTEDG0   ;make INT0 negative-edge
                              ;triggered
      BSF   INTCON,INT0IE     ;enable INT0
      MOVLW B'00100000';enable transmit and choose low baud
      MOVWF TXSTA             ;write to reg
      MOVLW D'15'       ;9600 bps (Fosc / (64 * Speed) - 1)
      MOVWF SPBRG             ;write to reg
      BCF   TRISC, TX   ;make TX pin of PORTC an output pin
      BSF   RCSTA, SPEN       ;enable the serial port
      BSF   INTCON,GIE        ;enable interrupts globally
;-- send control byte to enable write
      BSF   PORTC,RC2         ;enable the RTC
      CALL  SDELAY
      MOVLW 0x8F              ;control register
      CALL  SPI
      MOVLW 0x0               ;clear WP bit for write
      CALL  SPI
      BCF   PORTC,RC2         ;disable RTC
      CALL  SDELAY
;-- send the data
      BSF   PORTC,RC2   ;enable for multibyte write
      MOVLW 0x87              ;Alarm0 address
      CALL  SPI               ;send address
      MOVLW 0x08              ;alarm at 8 seconds
      CALL  SPI               ;send second
      MOVLW 0x80              ;once-per-minute
      CALL  SPI               ;send minute
      MOVLW 0x80              ;once-per-minute
      CALL  SPI               ;send hour
      MOVLW 0x80              ;once-per-minute
      CALL  SPI               ;send day
      BCF   PORTC,RC2         ;end of multibyte write
      CALL  SDELAY
```

626

```
;-- send control byte to enable INT0
      BSF   PORTC,RC2        ;enable the RTC
      CALL  SDELAY
      MOVLW 0x8F             ;control register of DS1306
      CALL  SPI
      MOVLW 0x01             ;enable INT0 pin of DS1306
      CALL  SPI
      BCF  PORTC,RC2         ;disable RTC
      CALL  SDELAY
LOOP  BRA LOOP               ;wait for interrupt
;-- service Alarm0
INT0_ISR
      BSF   PORTC,RC2        ;enable the RTC
      CALL  SDELAY
      MOVLW 0x8F             ;control register
      CALL  SPI
      MOVLW 0x04             ;1 Hz on, Alarm0 off
      CALL  SPI
      BCF  PORTC,RC2         ;disable RTC
      CALL  SDELAY
;-- send Alarm0 seconds to reset alarm
      BSF   PORTC,RC2        ;enable the RTC
      CALL  SDELAY
      MOVLW 0x87             ;Alarm0 seconds register
      CALL  SPI
      MOVLW 0x08             ;at 8 seconds
      CALL  SPI
      BCF  PORTC,RC2         ;disable RTC
      CALL  SDELAY
;-- begin displaying
      MOVLW upper(MESSAGE)
      MOVWF TBLPTRU
      MOVLW high(MESSAGE)
      MOVWF TBLPTRH
      MOVLW low(MESSAGE)
      MOVWF TBLPTRL
NEXT  TBLRD*+                ;read the characters
      MOVF  TABLAT,W         ;place it in WREG
      IORLW 0x0
      BZ    OVER             ;if end of line, start over
      CALL  TRANS            ;send char to serial port
      BRA   NEXT             ;repeat for the next character
;-- send control byte to enable INT0
OVER  BSF   PORTC,RC2        ;enable the RTC
      CALL  SDELAY
      MOVLW 0x8F             ;control register
      CALL  SPI
      MOVLW 0x01             ;1 Hz off, Alarm0 on
      CALL  SPI
      BCF  PORTC,RC2         ;disable RTC
      CALL  SDELAY
      BCF   INTCON,INT0IF
```

627

```
      RETFIE
;-- SPI subroutine
;-- serial data transfer subroutine
;-- delay for SPI communications
      RETURN;SEE PREVIOUS PROGRAMS FOR ABOVE SUBROUTINES
;--message to be displayed upon interrupt
MESSAGE:  DB   0x0A,0x0D,"Yes",0
      END
```

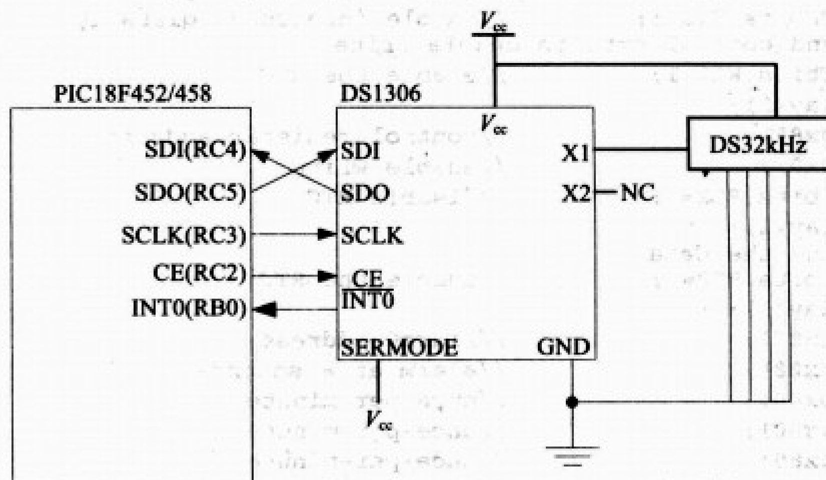


图 16-15 DS1306 通过硬件 INT0 连接到 PIC18

以下是上面代码的C语言版本。

程序 16-6C

```
//Program 16-6C
#include <p18f458.h>
//INSERT FUNCTION PROTOTYPES
#pragma interrupt chk_isr  //used for high priority inter-
rupt only
void chk_isr (void)
{
  if (INTCONbits.INT0IF==1)//INT0 caused interrupt?
  INT0_ISR( );             //Yes. Execute INT0 program
}
#pragma code My_HiPrio_Int=0x0008 //high-priority interrupt
void My_HiPrio_Int (void)
{
  _asm
  GOTO chk_isr
  _endasm
}
#pragma code
void main(void)
   {
//-- initialize SPI, INT0, and USART
  TRISC=0x90;      //make PORTC output, except SDI and RX
  TRISBbits.TRISB0=1;      //make RB0 input for interrupt
```

628

```
   SSPSTAT=0x0;              //middle read, active edge send
   SSPCON1=0x22;             //master SPI enable, Fosc / 64
   INTCON2bits.INTEDG0=0;    //make INT0 negative edge
                             //triggered
   INTCONbits.INT0IE=1;      //enable INT0
   TXSTA=0x20;           //enable transmit and choose low baud
   SPBRG=15;             //9600 bps (Fosc / (64 * Speed) - 1)
   RCSTAbits.SPEN=1;         //enable the serial port
   INTCONbits.GIE=1;         //enable interrupts globally
//-- send control byte to enable write
   PORTCbits.RC2=1;          //enable the RTC
   MSDelay(1);
   SPI(0x8F);                //control register address
   SPI(0x0);                 //enable write
   PORTCbits.RC2=0;          //disable RTC
   MSDelay(1);
//-- send the data
   PORTCbits.RC2=1;          //enable the RTC
   MSDelay(1);
   SPI(0x87);                //Alarm0 address
   SPI(0x08);                //alarm at 8 seconds
   SPI(0x80);                //once-per-minute
   SPI(0x80);                //once-per-minute
   SPI(0x80);                //once-per-minute
   PORTCbits.RC2=0;          //disable RTC
   MSDelay(1);
//-- send control byte to enable INT0
   PORTCbits.RC2=1;          //enable the RTC
   MSDelay(1);
   SPI(0x8F);                //control register
   SPI(0x01);                //enable INT0
   PORTCbits.RC2=0;          //disable RTC
   MSDelay(1);
   while(1);                 //wait for interrupt
   }
//-- service Alarm0
void INT0_ISR()
   {
   unsigned char mess[]={0x0D,0x0A,'Y','E','S',0};
   unsigned char i;
   PORTCbits.RC2=1;          //enable the RTC
   MSDelay(1);
   SPI(0x8F);                //control register
   SPI(0x04);                //1 Hz on, Alarm0 off
   PORTCbits.RC2=0;          //disable RTC
   MSDelay(1);
//-- send Alarm0 seconds to reset alarm
   PORTCbits.RC2=1;          //enable the RTC
   MSDelay(1);
   SPI(0x87);                //Alarm0 seconds register
   SPI(0x08);                //at 8 seconds
   PORTCbits.RC2=0;          //disable RTC
   MSDelay(1);
//-- begin sending the data
   for(i=0;mess[i]!=0;i++)
```

```
    TRANS(mess[i]);
//-- send control byte to enable INT0
   PORTCbits.RC2=1;            //enable the RTC
   MSDelay(1);
   SPI(0x8F);                  //control register
   SPI(0x01);                  //1 Hz offbits. Alarm0 on
   PORTCbits.RC2=0;            //turn off RTC
   INTCONbits.INT0IF=0;
   }
//--SEE PREVIOUS EXAMPLES FOR SUBROUTINES
```

在上面的程序中，是通过向串行端口发送信息来表明警报发生的。这里使用 32 kHz 的输出来驱动实际的报警器。尽管对于人类的耳朵来说，32 kHz 的频率显得太高，但是可以使用多个 *D* 触发器来降低频率，如图 16-16 所示。对于图 16-16，修改程序 16-6 的工作留给读者自行完成。

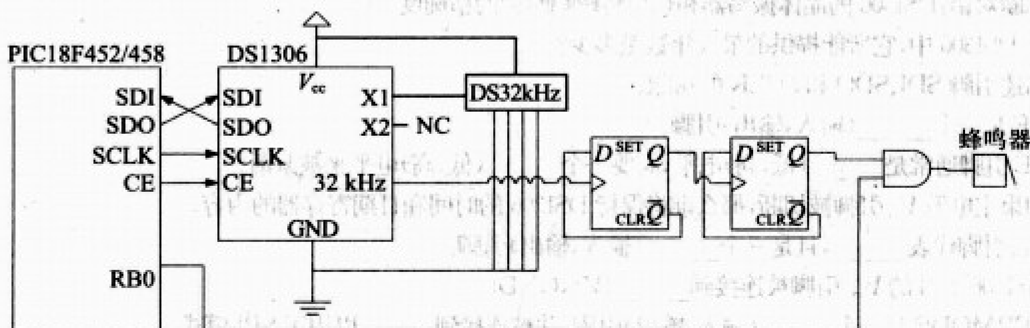


图 16-16　DS1306 连接到 PIC18，带有嗡鸣器控制

复习题

1. 控制寄存器的哪个位属于 1-Hz 引脚?
2. 判断对错:DS1306 的 INT0 引脚是一个输入引脚。
3. 判断对错:INT0 引脚是低电平有效。
4. 控制寄存器的哪个位属于警报 1 的中断?
5. 写出警报 1 的地址。

630

小结

本章首先描述了 SPI 总线的连接和协议，接着还讨论了 DS1306 RTC 芯片的每一个引脚的功能。DS1306 可被用来为许多应用提供实时时钟以及日期。此外，还讨论了 RTC 的各种特征，并给出了大量的编程实例。

习题

1. 判断对错:SPI 总线需要外部时钟。
2. 判断对错:SPI　CE 是低电平有效。

3. 判断对错:SPI 总线有一个单 D_{in} 引脚。
4. 判断对错:SPI 总线有多个 D_{out} 引脚。
5. 判断对错:当 SPI 设备被用作从设备时,SCLK 是一个输入引脚。
6. 判断对错:在 SPI 设备中,数据是以 8 位的块为单位来传输的。
7. 判断对错:在 SPI 设备中,信息(数据,地址)的每一位是以单时钟脉冲同步传输的。
8. 判断对错:在 SPI 设备中,8 位地址是跟在 8 位数据之后传输的。
9. 对于数据引脚,SPI 和三线连接之间有何区别?
10. SPI 协议如何区分读周期和写周期。
11. DS1306 DIP 封装是一个______脚的封装。
12. 哪一个引脚被分配为主要的 V_{cc}?
13. 在 DS1306 中有多少个引脚被指定为地址/数据引脚?
14. 判断对错:DS1306 需要一个外部的晶体振荡器。
15. 判断对错:DS1306 的晶体振荡器和热量将影响时钟的精确度。
16. 在 DS1306 中,它所能提供的最大年数是多少?
17. 描述引脚 SDI、SDO 和 SCLK 的功能。
18. CE 是一个______(输入,输出)引脚。
19. CE 引脚通常是______(低,高)电平,需要一个______(低,高)电平来被激活。
20. 如果主电源 V_{cc} 引脚被切断,那么谁将保持 DS1306 的时间和日期寄存器的内容。
21. V_{bat} 引脚代表______,且是一个______(输入,输出)引脚。
22. DS1306 芯片的 V_{cc2} 引脚被连接到______(V_{cc},GND)。
23. SERMODE 是一个______(输入,输出)引脚,并被连接到______以用于 SPI 模式。
24. V_{cc1} 是一个______(输入,输出)引脚,并被连接到______电压。
25. 1-Hz 是一个______(输入,输出)引脚。 631
26. INT0 是一个______(输入,输出)引脚。
27. 32 kHz 是一个______(输入,输出)引脚。
28. INT1 是一个______(输入,输出)引脚。
29. DS1306 总共有______字节的地址,请给出读和写操作的地址。
30. 如果 V_{cc} 引脚的电源丢失,DS1306 的时间和数据寄存器的内容是什么?
31. 如果 V_{cc} 引脚的电源丢失,通用目的 RAM 区域的内容是什么?
32. 在什么时候 DS1306 将切换电池供电模式?
33. 赋给实时时钟寄存器的地址是什么?
34. 赋给日历的地址是什么?
35. 哪一个寄存器被用来设置 AM/PM 模式,请给出该寄存器的位地址。
36. 哪一个寄存器被用来设置 24 小时模式,请给出该寄存器的位地址。
37. DS1306 在存储器的哪个位置存储 2007 年?
38. DS1306 的 RAM 的最后区域的地址是什么?
39. 判断对错:DS1306 只提供 BCD 形式的数据。
40. 编制程序,获取 BCD 形式的年数据,并将它送到端口 B 和端口 D。
41. 编制程序,获取 BCD 形式的小时和分钟数据,并将它送到端口 B 和端口 D。
42. 编制程序,设置时间为 9:10:05。
43. 编制程序,设置时间为 22:47:19。

44. 编制程序,设置日期为2009年5月14日。
45. V_{bat}和V_{cc2}的作用是什么?
46. 编制C程序,显示AM/PM模式的时间。
47. 编写C程序,获得BCD形式的年数据,并将它送到端口*B*和端口*D*。
48. 编写C程序,获取BCD形式的小时和分钟数据,并将它送到端口*B*和端口*D*。
49. 编写C程序,设置时间为9:10:05。
50. 编写C程序,设置时间为22:47:19。
51. 编写C程序,设置日期为2009年5月14日。
52. 在第51题中,RTC如何跟踪一个世纪?
53. INT0是一个______(输入,输出)引脚,______(高,低)电平有效。
54. 1-Hz是一个______(输入,输出)引脚。
55. 请给出属于警报中断的控制寄存器的位地址以及如何使能它。
56. 请给出属于1-Hz引脚的控制寄存器的位地址以及使能它。
57. 请给出属于警报0中断的控制寄存器的位地址以及如何使能它。
58. 请给出属于警报1中断的控制寄存器的位地址以及如何使能它。
59. 判断对错:对于32 kHz的输出引脚,频率是设定的,不能被改变。
60. 给出能激活INT1引脚的中断源。
61. 为什么要将AIE0指向IRQ引脚?
62. IROF0和AIE0位的区别是什么?
63. IROF1和AIE1位的区别是什么?
64. 如何使1-Hz引脚输出方波。
65. 哪一个寄存器被用于设置每秒响一次的警报1?
66. 如果选择了警报每分钟响一次,请解释IRQ1F引脚是如何被激活的。

632

复习题答案

16.1节

1. 正确。 2. 正确。 3. 错误。 4. 错误。
5. 在单字节模式下,在每个字节后且在新的周期之前,必须将CE引脚置为低电平。在成组模式下,对于成组(多字节)传输期间,必须保持CE引脚为高电平。

16.2节

1. 正确。前提是只有V_{bat}被连接到外部电池时。
2. 7 3. 96 4. 正确。 5. 引脚11是SCLK。 6. 错误。SERMODE=V_{cc}。

16.3节

1. 正确。 2. 0~6 3. 20~7FH 4. 正确。 5. 错误。 6. 错误。

16.4节

1. 第2位。 2. 错误。 3. 正确。 4. 第1位。
5. 字节地址0B~0E(十六进制)用于读操作,而字节地址8B~8E(十六进制)用于写操作。

633

第17章 电机控制：继电器、PWM、DC电机和步进电机

学习目标：

- ☐继电器的基本操作
- ☐继电器和PIC18的接口
- ☐光隔离器的基本操作
- ☐光隔离器和PIC18的接口
- ☐步进电机的基本操作
- ☐步进电机和PIC18的接口
- ☐控制和操作步进电机的PIC18编程
- ☐有关步进电机定义的术语
- ☐DC电机的基本操作
- ☐DC电机与PIC18的接口
- ☐控制和操作DC电机的PIC18编程
- ☐电机速度的PWM控制
- ☐控制和操作DC电机的CCP编程
- ☐控制和操作DC电机的ECCP编程

本章将讨论电机的控制，以及PIC18与继电器、光隔离器、步进电机和DC电机的接口。17.1节描述继电器和光隔离器的基本知识，然后讲述它们与PIC18的接口。17.2节介绍步进电机和PIC18的接口，17.3节讲述DC电机的特点以及它与PIC18的接口。此外，还讨论了PWM。17.4节将使用PIC18的CCP特征来控制DC电机。17.5节介绍ECCP在电机控制中的使用。在编程实例中，均给出了汇编语言和C语言两个版本。

17.1 继电器和光隔离器

本节将首先回顾机电继电器、固态继电器、簧片开关和光隔离器的基本操作，然后描述它们与PIC18的接口。本节将同时使用汇编语言和C语言编程来说明如何控制它们。

17.1.1 机电继电器

继电器是一种广泛应用于工业控制、自动化和电器设备的电气控制开关。它可以隔离一个系统中具有不同电压源的两部分。比如，一个+5 V的系统可通过一个继电器与另一个

+120 V的系统实现隔离。其中有一种继电器叫作电机或者电磁继电器(EMR),如图 17-1 所示。EMR 由 3 个部分组成:线圈、弹簧和触头。在图 17-1 中,左边的+5 V 数字电压源能够控制右边的 12 V 电机,但是在它们之间没有任何的物理接触。当电流流过线圈时,在线圈周围将产生磁场(线圈被磁化),这将使得衔铁被线圈吸引。衔铁的触头就像一个开关一样,可以闭合和断开电路。当线圈未被磁化时,弹簧将衔铁拉到它正常开或者关的状态。在 EMR 的框图中,并没有画出弹簧,但是它确实存在于继电器内部。对于各种不同的应用,也有不同类型的继电器。在选择一个继电器时,需要考虑以下几点。

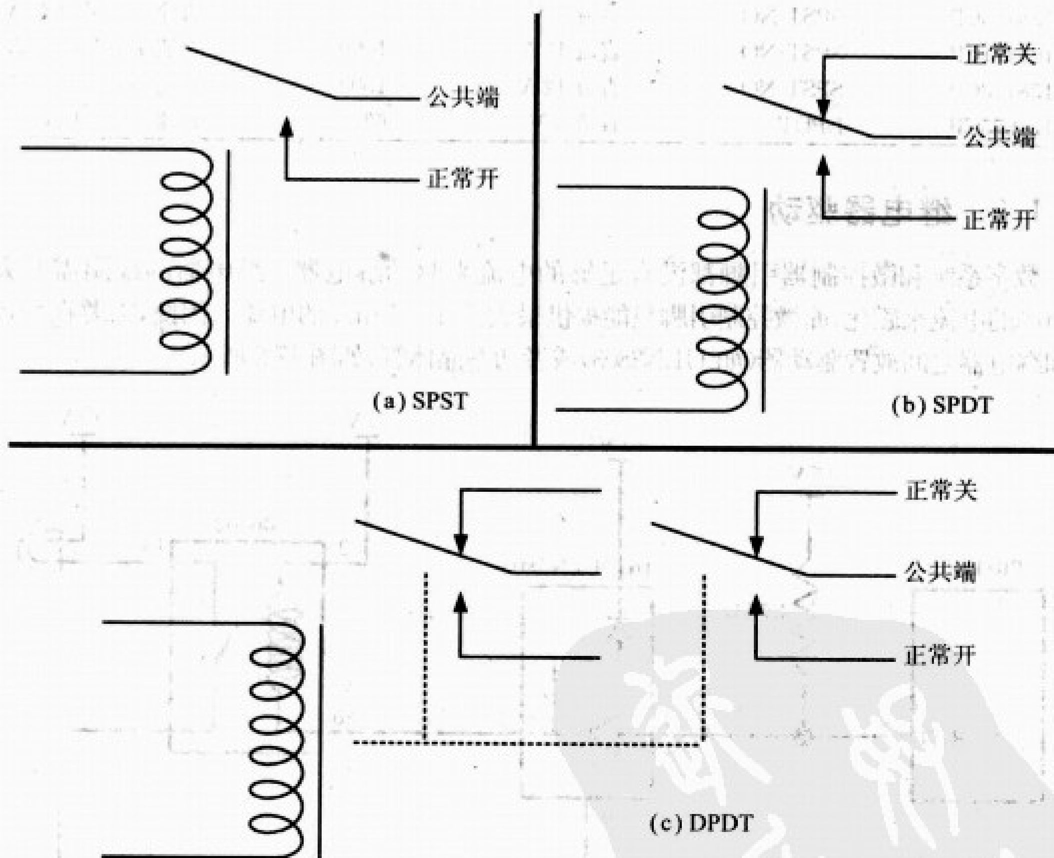


图 17-1 继电器的示意图

(1) 触头可以是正常的开(NO)或者正常的关(NC)。在 NC 类型中,当线圈未被磁化时,触头是闭合的;而在 NO 类型中,当线圈未被磁化时,触头是断开的。

(2) 可以有一个或者更多触头。比如,有 SPST(单刀单掷)、SPDT(单刀双掷)和 DPDT(双刀双掷)继电器。

(3) 需要使用电压和电流源来磁化线圈。电压可以是从几伏到 50 V 不等,而电流可以是从几毫安到 20 mA 不等。继电器有一个最小电压,低于这个电压时线圈将不能被磁化。这个最小电压被称作"牵引"电压。在继电器的数据表中,可能看到的不是电流,而是线圈的阻抗。比如,如果线圈电压是 5 V,线圈的阻抗是 500 Ω,那么就需要一个最小为 10 mA(5 V/500 Ω=10 mA)的牵引电流。

(4) 触头可承受的最大 DC/AC 电压和电流。电压可以是从几伏到几百伏不等,而电流可以是从几安培到 40 A 或更多,视不同继电器而定。注意这里的标称电压/电流和用于线圈磁化的电压/电流之间的区别。在一边使用小电压/电流来控制另一边的大电压/电流,这就是继电器在工业控制中得到如此广泛应用的原因。表 17-1 给出了一些继电器的特征。

表 17-1 典型的 DIP 继电器的特征(www.Jameco.com)

型 号	触点形式	线圈电压	线圈电阻(Ω)	触点电压-电流
106462CP	SPST-NO	直流 5 V	500	直流 100 V,0.5 A
138430CP	SPST-NO	直流 5 V	500	直流 100 V,0.5 A
106471CP	SPST-NO	直流 12 V	1000	直流 100 V,0.5 A
138448CP	SPST-NO	直流 12 V	1000	直流 100 V,0.5 A
129875CP	DPDT	直流 5 V	62.5	直流 30 V,1 A

637

17.1.2 继电器驱动

数字系统和微控制器引脚都没有足够的电流来驱动继电器。继电器的线圈需要大概 10 mA的电流来磁化,而微控制引脚只能提供最大为 1~2 mA 的电流。因此,需要在微控制器和继电器之间放置驱动器(如 ULN2803)或者功率晶体管,如图 17-2 所示。

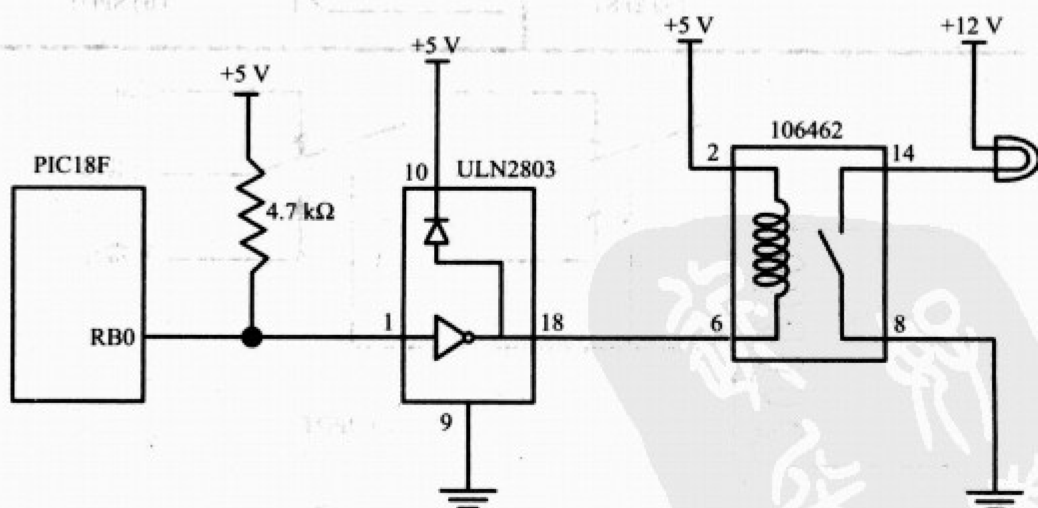


图 17-2 PIC18 与继电器的连接

程序 17-1 通过每隔几毫秒磁化和去磁化继电器一次来打开或者关闭电灯,如图 17-2 所示。

程序 17-1

```
;Program 17-1
R3    SET   0x20        ;set aside location 0x20 for R3
R4    SET   0x21        ;loc. 0x21 for R4
      ORG 0H
      BCF TRISB,0       ;PORTB.0 as output
OVER  BSF PORTB,0       ;turn on the lamp
```

```
        CALL DELAY
        BCF   PORTB,0     ;turn off the lamp
        CALL DELAY
        BRA OVER
DELAY   MOVLW 0xFF
        MOVWF R4
D1      MOVLW 0xFF
        MOVWF R3
D2      NOP
        NOP
        DECF  R3,F
        BNZ   D2
        DECF  R4,F
        BNZ   D1
        RETURN
```

638

17.1.3 固态继电器

另一个广泛应用的继电器就是固态继电器，如表 17-2 所示。这个继电器没有线圈、弹簧或者机械的触头开关。整个继电器由半导体材料制成。由于固态继电器中没有机械部分，它们的开关响应时间要比机电继电器快得多。固态继电器的另一个优点在于它有更长的使用寿命。机电继电器的使用寿命只有几万次到几百万次不等。触头点的磨损和拉伸可能导致继电器在短时间内工作异常，而固态继电器就没有这样的局限。极小的输入电流和封装使得固态继电器成为微处理器和逻辑开关的理想选择。它们被广泛地用来控制泵、螺线管、报警器和其他功率器件。一些固态继电器还有相位控制功能，这是电机速度控制和灯光控制应用的理想选择。图 17-3 给出了使用固态继电器(SSR)来控制风扇的一个例子。

表 17-2 部分固态继电器特性(www.Jameco.com)

型号	触点形式	控制电压	触点电压	触点电流
143058CP	SPST	直流 4～32 V	直流 240 V	直流 3 A
139053CP	SPST	直流 3～32 V	直流 240 V	直流 25 A
162341CP	SPST	直流 3～32 V	直流 240 V	直流 10 A
172591CP	SPST	直流 3～32 V	直流 60 V	直流 2 A
175222CP	SPST	直流 3～32 V	直流 60 V	直流 4 A
176647CP	SPST	直流 3～32 V	直流 120 V	直流 5 A

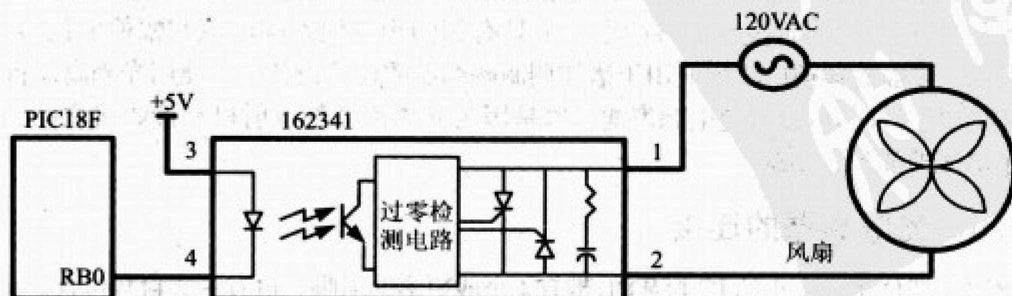


图 17-3 PIC18 与固态继电器的连接

639

17.1.4 簧片开关

另一个流行的开关是簧片开关。当簧片开关被放置在磁场中时,触头闭合。当磁场被去掉时,触头被弹簧撑开,如图17-4所示。簧片开关是用于湿地和海洋环境的理想选择,因为它们可以沉浸在燃料或水中。簧片开关严密的封装使得它们被广泛地应用在肮脏而多尘的环境中。

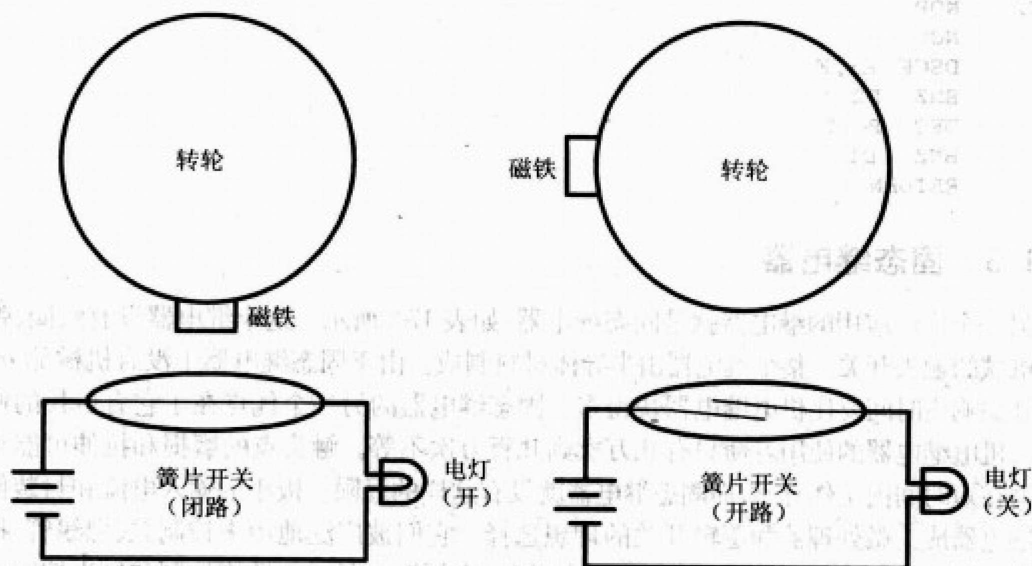


图17-4 簧片开关和磁铁的组合

17.1.5 光隔离器

在一些设备中,也使用光隔离器(或者叫光耦合器)来隔离一个系统的两部分。其中一例就是电机驱动。电机可以产生反向EMF,这是由于电路的突然变化而产生的高电压浪涌,如公式 $V=Ldi/dt$ 所示。在诸如印制电路板设计的情形中,可以使用去耦电容(请参阅附录C)来降低电压浪涌的影响。在带有电感(线圈绕组)的系统(如电机)中,使用去耦电容或者二极管不起作用,在这种情况下可以使用光隔离器。光隔离器有一个LED(发光二极管)发射机和光敏传感接收机,两者由一个间隙来隔开。当电流流过二极管时,它通过这个间隙发射光信号,而与此同时接收机将产生一个具有相同相位但不同电流和幅值的信号,如图17-5所示。光隔离器被广泛地应用于诸如调制解调器的通信设备中。使用光隔离器将计算机连接到电话线时,可以避免电源浪涌。这是因为光隔离器的发射机和接收机之间的间隙阻止了电流浪涌到达系统。

640

17.1.6 光隔离器的连接

光隔离器位于一个小的IC封装内,带有4个或更多的引脚。也有一些封装包含一个以上的光隔离器。当将一个光隔离器放置在两个电路之间时,必须使用两个独立的电压源,如图17-6所示。与继电器不同的是,在微控制器/数字输出和光隔离器之间是不需要有驱动器的。

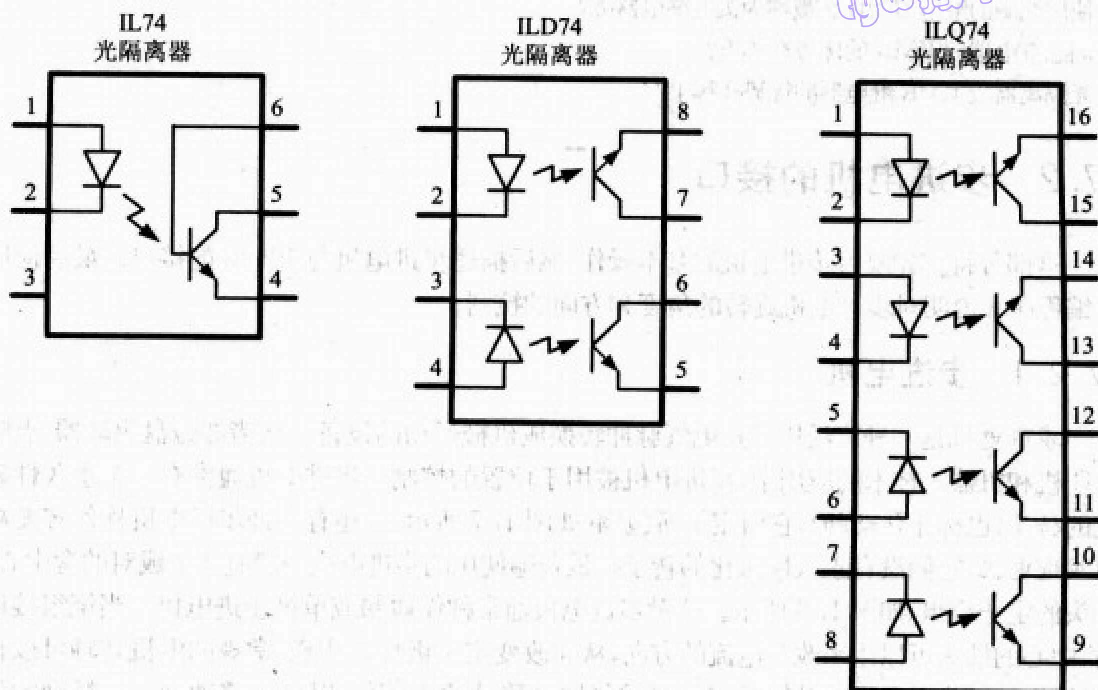


图 17-5 光耦合器封装举例

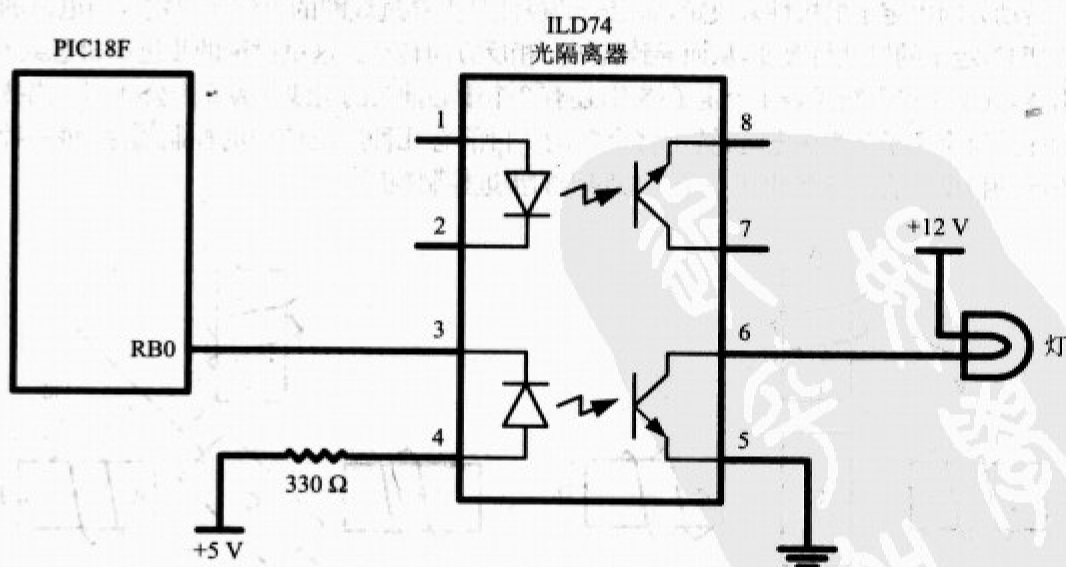


图 17-6 通过光耦合器控制电灯

641

17.1.7 复习题

1. 请指出继电器的一个应用。
2. 为什么要在微控制器和继电器之间放置驱动器呢?
3. 什么是 NC 继电器?

4. 使用线圈的继电器为什么被称为机电继电器呢?
5. 固态继电器较 EMR 的优势在哪里?
6. 光隔离器较 EMR 继电器的优势在哪里?

17.2 步进电机的接口

这部分将首先复习步进电机的基本操作,然后描述步进电机与 PIC18 的接口。最后使用汇编程序来说明对步进电机旋转的角度和方向的控制。

17.2.1 步进电机

步进电机是一种广泛用于将电气脉冲转换成机械运动的设备。在诸如磁盘驱动器、点阵打印机和机器人技术的应用中,步进电机被用于位置的控制。步进电机通常有一个永久性磁化的转子(也称作传动轴),它由定子所包围(如图 17-7 所示)。还有一种步进电机称作可变磁阻步进电机,它们没有永久性磁化的转子。最普遍使用的步进电机一般有 4 个成对的含中心抽头的定子绕组,如图 17-8 所示。这种步进电机通常称作四相或单极步进电机。当绕组接地时,中心的抽头可以用来改变电流的方向,从而改变定子极性。注意,常规的电机转轴可以自由地转动,而步进电机的转轴是定值可重复增量移动的,从而可以将它移动到一个精确的位置。这种可重复的定值运动是可能的,这基于一个基本的磁场理论:同极相斥,异极相吸。转子的转动方向由定子的极性来决定,而定子的极性又由穿过线圈的电流所决定。当电流的方向改变时,定子的极性将改变,从而导致转子向相反方向转动。这里讨论的步进电机总共有 6 根引线,其中 4 根引线代表 4 个定子绕组,还有 2 个中心抽头的引线代表 2 个公共端。当按一定顺序给每个定子绕组供电时,转子将会转动。目前有几种广泛使用的控制顺序,每一种都有不同的精度。表 17-3 给出了一个二相四步的步进控制顺序。

642

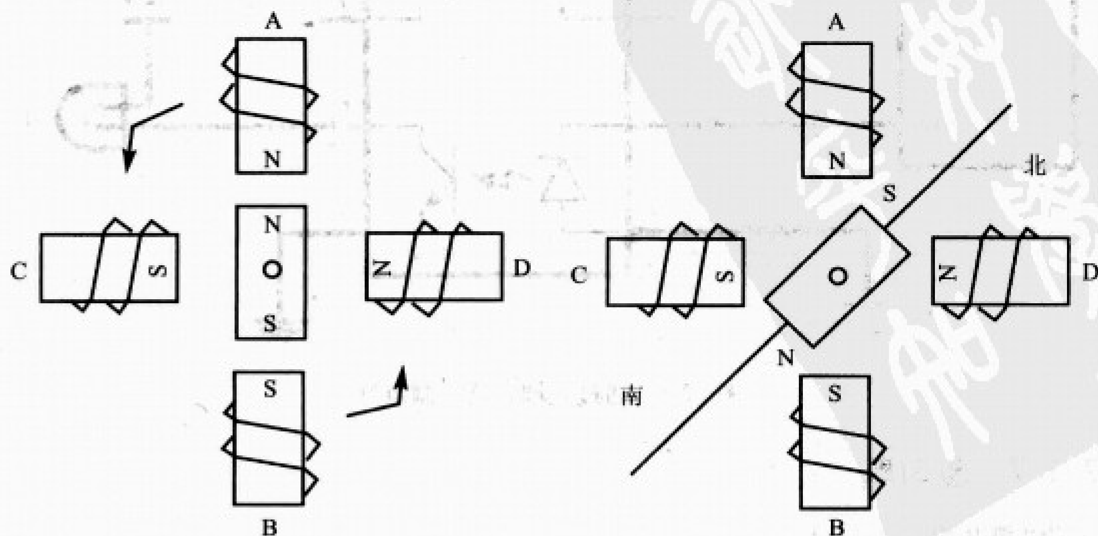


图 17-7 转子排列

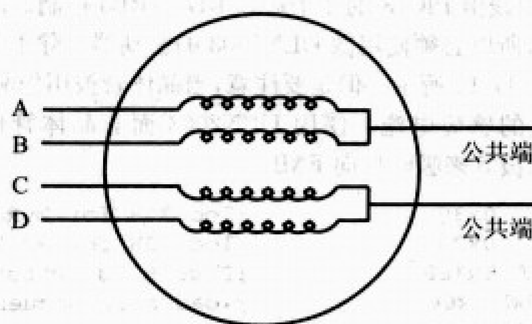


图 17-8 定子绕组的配置

注意，尽管可以从表 17-3 中的任何顺序开始，但是一旦开始，就必须以正确的顺序继续运行下去。比如说，如果从第 3 步 0110 开始，必须按 4、1、2 这个顺序来进行。

表 17-3 正常的四步控制顺序

顺时针 ↓ 逆时针 ↑

步的序号	绕组 A	绕组 B	绕组 C	绕组 D
1	1	0	0	1
2	1	1	0	0
3	0	1	1	0
4	0	0	1	1

17.2.2 步进角

执行一个单步，到底运动量有多少呢？这取决于电机的内部结构，特别是定子和转子上的齿数。步进角是执行一个单步的最小旋转角度。不同的电机有不同的步进角。表 17-4 列举一些电机的步进角。在表 17-4 中，请留意“每转一圈所需的步数”。这是需要完成一个完整的旋转（或者 360°）所需要的总的步数（例如 180 步×2°=360）。

表 17-4 步进电机的步进角

步进角	每转一圈所需的步数
0.72	500
1.8	200
2.0	180
2.5	144
5.0	72
7.5	48
15	24

必须注意的是，步进电机的定子不需要更多的终端引线来获得更小的步距。对于本节中讨论的所有步进电机，使用 4 个引线用于定子线圈，而使用 2 个公共引线用于中间抽头。尽管一些制造商只有 1 个引线用于公共信号端，而不是通常的 2 个引线，但总是有 4 根引线是用于定子的，如例 17-1 所示。下面将介绍一些相关的术语使你更深入地理解步进电机。 643

例 17-1 描述图 17-9 中的步进电机和 PIC18 的连接，并编制程序让步进电机连续地转动。

解：

使用以下步骤说明步进电机和 PIC18 的接口及其编程。

（1）使用欧姆表测量引线的电阻。这样就可以识别哪个公共引线和哪个绕组引线是相连接的。

（2）将公共线缆连接到电机电源的正极。在许多电机中，+5V 就足够。

(3) 定子线圈的4根引线由PIC18的4个端口RB0～RB3控制。由于PIC18没有足够大的电流来驱动步进电机线圈，所以必须使用像ULN2003的驱动器。除了ULN2003之外，还可以使用晶体管作为驱动器，如图17-11所示。但是要注意，当晶体管被用作驱动器时，必须使用二极管来吸收线圈被关闭时产生的感应电流。使用ULN2003而非晶体管作为驱动器的原因在于，ULN2003有一个内置的二极管来吸收反向EMF。

```
MyReg       SET   0x30              ;loc 30H for MyReg
R2          SET   0x20              ;loc 20H for R2 Reg
            CLRF  TRISB             ;Port B as output
            MOVLW  0x66             ;load step sequence
            MOVWF  MyReg
BACK        MOVFF  MyReg,PORTB      ;issue sequence to motor
            RRNCF  MyReg,F          ;rotate right clockwise
            CALL  DELAY             ;wait
            BRA   BACK              ;keep going
DELAY
            MOVLW  0xFF
            MOVWF  R2
D1          NOP
            DECF  R2,F
            BNZ  D1
            RETURN
            END
```

改变DELAY的值可以设置转动的速度。

644　在这里，也可以使用位操作指令BSF和BCF来代替RRNCF，以产生控制序列。

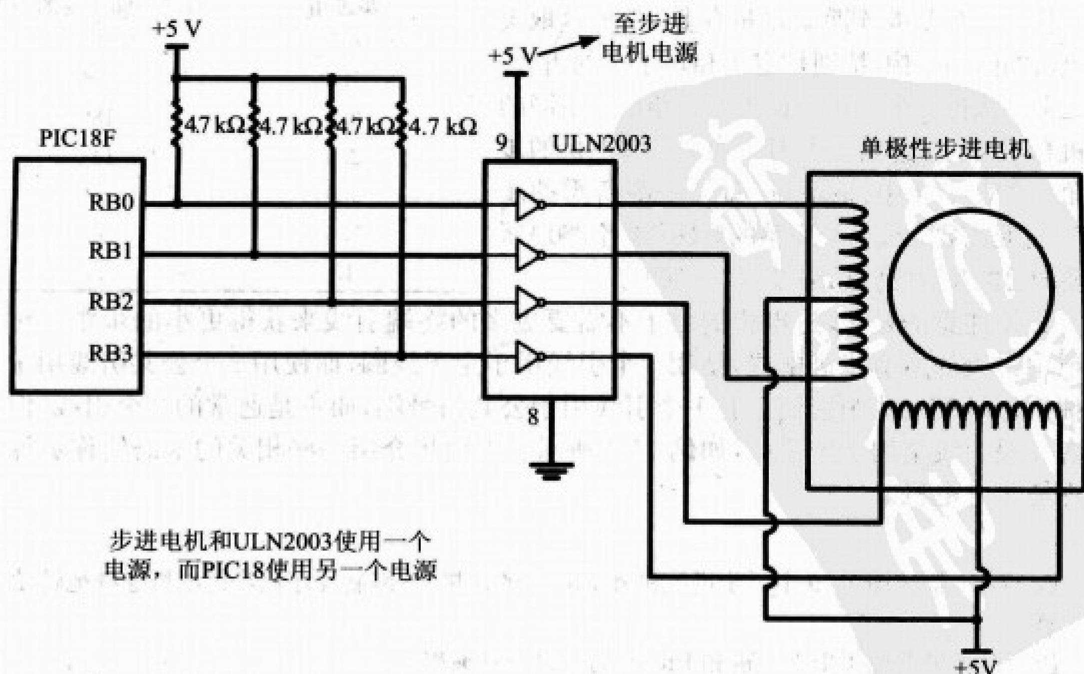


图17-9　PIC18与步进电机的接口

17.2.3 每秒的步数和 rpm 的关系

rpm(每分钟所转的圈数)、每圈所需的步数以及每秒所走的步数之间的关系如下：

$$每秒所走的步数=\frac{(rpm\times 每圈所需的步数)}{60}$$

17.2.4 四步顺序和电机转子的齿数

在前面表 17-3 中给出的切换顺序称作四步开关顺序，因为在执行 4 步之后，相同的两个线圈又会重现。那么与这 4 步相关的运动量有多少呢？在完成这 4 步之后，转子只移动一个齿距。因此在每圈 200 步的步进电机中，转子将有 50 个齿，因为需要4×50=200 步完成一圈。这里可以得出一个结论：最小的步进角总是转子齿数的一个函数。也就是说，步进角越小，转子转动的齿数越多，如例 17-2 所示。

例 17-2 假定步进电机有一个 2°的步进角。为了将步进电机移动 80°，使用表 17-3 所示的四步顺序。请指出需要使用多少次四步顺序。

解：

具有 2°步进角的步进电机有如下的特性：

步进角：2°　　　　每转一圈所需的步数：180

转子齿数：45　　　　每四步顺序对应的运动：8°

因此，为了将转子移动 80°，需要发送 10 个连续的四步脉冲序列，因为 10×4 步×2°=80°。

645

看到例 17-2，有人可能会想：如果需要移动 45°，怎么办呢？因为每步是移动 2°。为了较好地解决这一问题，所有的步进电机都允许使用八步开关顺序。这个八步顺序也称作*半程步进*，因为在八步顺序中，每步的步进角是正常步进角的一半。比如，当使用表 17-5 中的序列时，一个 2°步进角的电机就可以使用 1°的步进角。

表 17-5 半程步进的八步顺序

顺时针 ↓　　　　逆时针 ↑

步的序号	绕组 A	绕组 B	绕组 C	绕组 D
1	1	0	0	1
2	1	0	0	0
3	1	1	0	0
4	0	1	0	0
5	0	1	1	0
6	0	0	1	0
7	0	0	1	1
8	0	0	0	1

17.2.5 电机速度

电机速度是以每秒的步数(步/秒)来衡量的，它是开关速率的函数。注意在例 17-1中，通过改变时间延迟循环的长度，可以得到不同的转速。

17.2.6 保持转矩

以下是关于保持转矩的一个定义："当电机传动轴静止或 rpm 为 0 时，将传动轴从它的保持位置脱离所需要的、来自外部源的转矩的大小。这可以通过施加在电机上的额定电压和电流来测定。"转矩的度量单位是盎司-英寸(或千克-厘米)。

17.2.7 波驱动四步顺序

除了前面讨论的八步和四步顺序外，还有另外的一种称作波驱动四步顺序的控制顺序，如表 17-6 所示。注意，表 17-5 中的八步顺序是表 17-6 中的波驱动四步顺序和表 17-3中的正常四步顺序的简单组合。有关波驱动四步顺序的实验将留给读者自己完成。

表 17-6 波驱动四步顺序

顺时针 ↓ 逆时针 ↑

步的序号	绕组 A	绕组 B	绕组 C	绕组 D
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1

646

17.2.8 单极性与双极性步进电机的接口

有 3 种类型的步进电机接口：通用的、单极性的和双极性的。它们可以通过与电机的连接数量来区分。一个通用的步进电机有 8 个连接，而单极性步进电机有 6 个，双极性步进电机则有 4 个。通用的步进电机可以被配置为 3 种模式，而单极性步进电机只可以设置成单极性的或者双极性的。很明显，双极性的电机不能被设置为通用的或者单极性的模式。表 17-7 给出了步进电机的一些特征。图 17-10 画出了所有 3 种配置类型基本的内部连接。

表 17-7 部分步进电机的特性(www.Jameco.com)

型号	步进角	驱动系统	电压	相电阻	电流
151861CP	7.5	单极性	5 V	9 Ω	550 mA
171601CP	3.6	单极性	7 V	20 Ω	350 mA
164056CP	7.5	双极性	5 V	6 Ω	800 mA

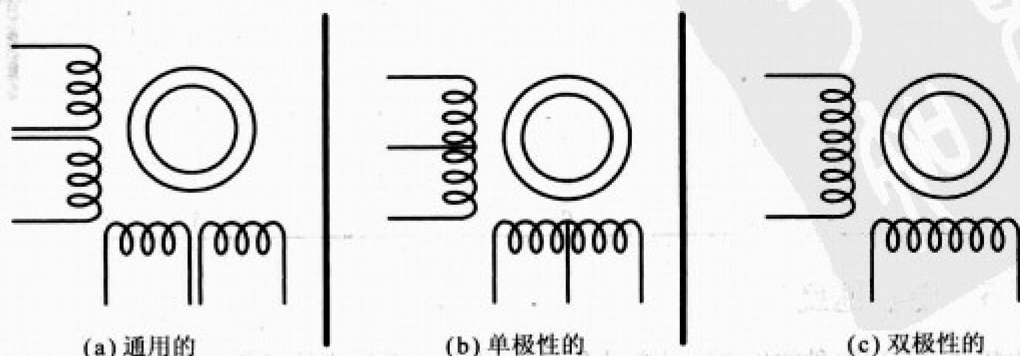


图 17-10 常用的步进电机类型

单极性步进电机可以使用如图 17-11 所示的基本连接来控制，而双极性步进电机需要 H-桥型的电路来控制。双极性步进电机需要比单极性电机更高的工作电流；但其好处在于它将获得更大的保持转矩。

17.2.9 使用晶体管作为驱动器

图 17-11 给出了晶体管与单极性步进电机的连接。其中，二极管用来遏制在线圈被磁化或者去磁化时所产生的反向 EMF 浪涌，这类似于前面讨论过的机电继电器。TIP 晶体管用来向电机提供更大的电流。表 17-8 列出了常见的工业用达林顿晶体管。这些晶体管可以承受更大的电压和电流。

647

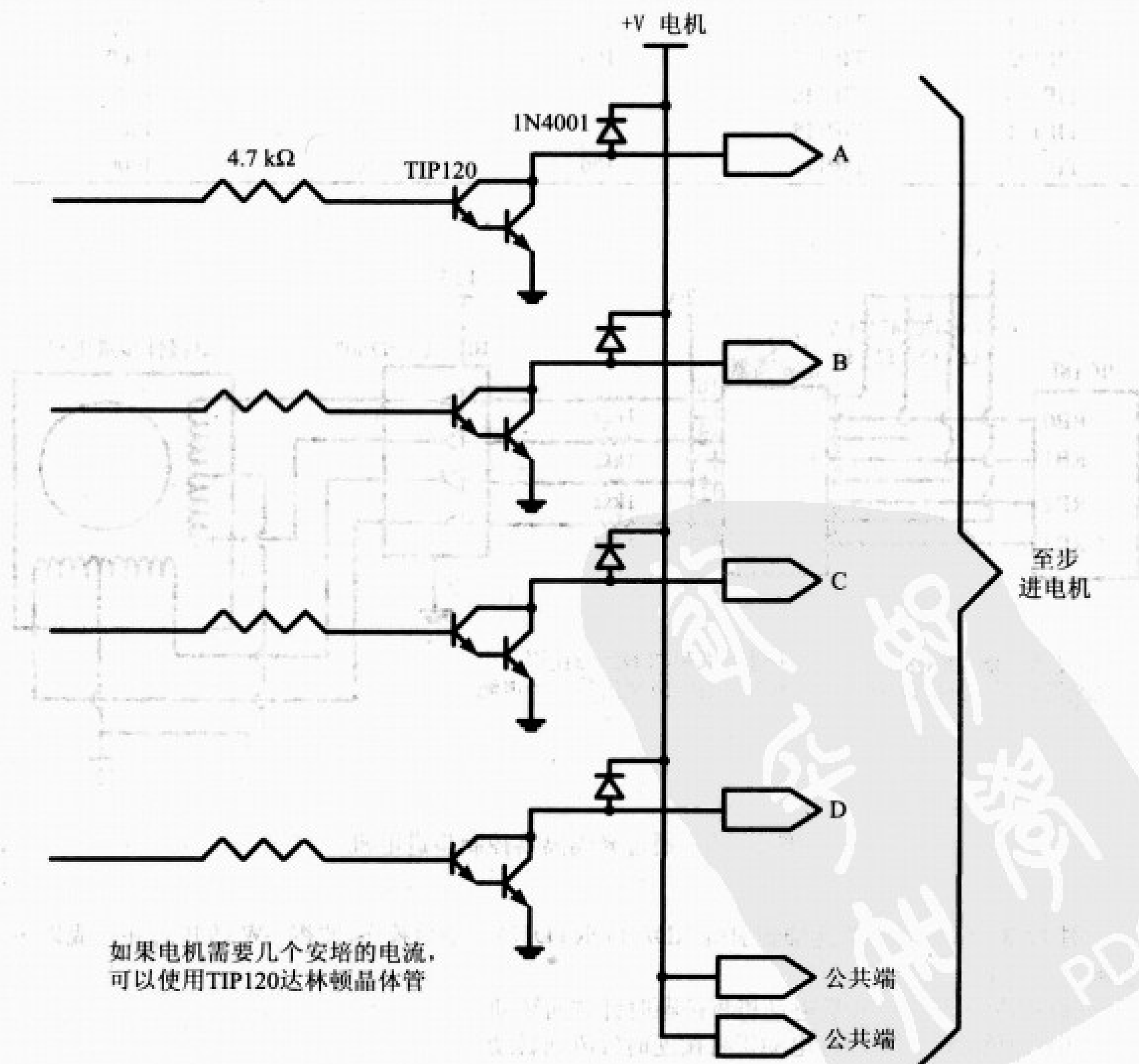


图 17-11 使用晶体管作为步进电机驱动器

648

17.2.10 通过光隔离器来控制步进电机

在17.1节中学习了光隔离器及其应用。光隔离器被广泛地用来隔离步进电机中的EMF电压，并保护数字/微控制器系统。如图17-12所示。并请参阅例17-3和例17-4。

表17-8 达林顿晶体管列表

NPN	PNP	V_{ceo}(V)	I_c(A)	hfe(公共端)
TIP110	TIP115	60	2	1000
TIP111	TIP116	80	2	1000
TIP112	TIP117	100	2	1000
TIP120	TIP125	60	5	1000
TIP121	TIP126	80	5	1000
TIP122	TIP127	100	5	1000
TIP140	TIP145	60	10	1000
TIP141	TIP146	80	10	1000
TIP142	TIP147	100	10	1000

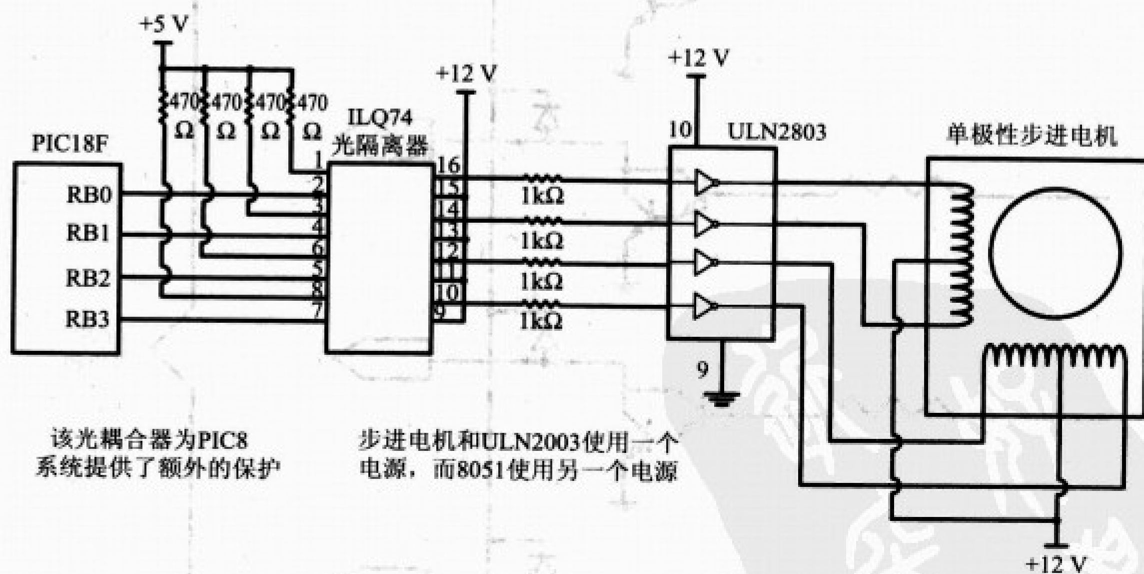


图17-12 通过光隔离器控制步进电机

例17-3 将一个开关连接到引脚RD7(PORTD.7)。编写程序，监视SW的状态并完成以下要求。

(a) 如果SW=0，步进电动机将按顺时针方向转动。

(b) 如果SW=1，步进电动机将按逆时针方向转动。

解：

```
MyReg     SET   0x30           ;loc 30H for MyReg
          BSF   TRISD,RD7      ;RD7 as input pin
          CLRF  TRISB          ;Port B as output
          MOVLW 0x66           ;load step sequence
          MOVWF MyReg
```

```
BACK    BTFSS PORTD,RD7      ;check the SW
        BRA   OVER      ;It is high. Make it clockwise
        MOVFF MyReg,PORTB    ;issue sequence to motor
        RRNCF MyReg,F        ;rotate right clockwise
        CALL  DELAY          ;wait
        BRA   BACK           ;keep going
OVER    MOVFF MyReg,PORTB    ;issue sequence to motor
        RLNCF MyReg,F        ;rotate left clockwise
        CALL  DELAY          ;wait
        BRA   BACK           ;keep going
```


17.2.11 用 PIC18 C 语言来控制步进电机

以下给出了 PIC18 控制步进电机的 C 语言版本。在这个程序中用到了≪(左移)和≫(右移)操作,这是在第 7 章中已学过的。

```
#include <p18f458.h>
void main()
  {
    TRISB=0x0;          //PORTB as output
    while(1)
      {
        PORTB = 0x66;
        MSDelay(100);
        PORTB = 0xCC;
        MSDelay(100);
        PORTB = 0x99;
        MSDelay(100);
        PORTB = 0x33;
        MSDelay(100);
      }
  }
```

例 17-4 将一个开关连接到引脚 RD7 (PORTD.7)。编制 C 程序,监视 SW 的状态并完成以下要求。

(a) 如果 SW=0 ,步进电动机将按顺时针方向转动。

(b) 如果 SW=1 ,步进电动机将按逆时针方向转动。

解:

```
#include <p18f458.h>
#define SW PORTDbits.RD7
void MSDelay(int ms);
void main()
  {
    TRISD=0x80;        //RD7 as input pin
    TRISB=0x0;         //PORTB as output
    while(1)
      {
        if(SW == 0)
          {
          PORTB = 0x66;
          MSDelay(100);
          PORTB = 0xCC;
          MSDelay(100);
          PORTB = 0x99;
          MSDelay(100);
          PORTB = 0x33;
```

650

```
            MSDelay(100);
            }
          else
            {
            PORTB = 0x66;
            MSDelay(100);
            PORTB = 0x33;
            MSDelay(100);
            PORTB = 0x99;
            MSDelay(100);
            PORTB = 0xCC;
            MSDelay(100);
            }
        }
    }

void MSDelay(unsigned int value)
  {
    unsigned int x, y;
    for(x=0;x<1275;x++)
      for(y=0;y<value;y++);
  }
```

17.2.12 复习题

1. 如果从0110开始，请给出步进电机的四步顺序。
2. 一个步进电机步进角为5°，那么每转一圈需要 ________ 步。
3. 为什么要在微控制器和步进电机之间放置一个驱动器？

17.3 DC电机的接口和PWM

在这一节，首先回顾DC电机的基本操作，然后介绍DC电机与PIC18的接口。最后，使用汇编和C语言程序来说明脉冲宽度调制(PWM)的概念，并展示如何控制DC电机的速度和方向。

17.3.1 DC电机

直流(DC)电机是又一个被广泛使用的将电脉冲转换为机械运动的设备。在DC电机中，只有"+"和"−"的引线。将它们连接到DC电压源，就能使电机向着一个方向转动。通过改变极性，DC电机将向相反的方向转动。DC电机的相关实验是非常简单的。比如，在许多主板上，用于CPU散热的小风扇是由DC电机驱动的。将其引线连接到"+"和"−"的电压源上，DC电机就可以开始运作。步进电机以1～15°的步数转动，而DC电机则可以连续地转动。在步进电机中，如果知道了初始位置，就可以轻易地计算出电机转动的步数以及电机的最终位置，而在DC电机中这是不可能的。DC电机的最大转速使用rpm来表示，并将在数据表中给出。DC电机有两种rpm:空载时的rpm和负载时的rpm 。制造商的数据表给出的是空载时的rpm。空载时的rpm可以是几千乃至几万。当有负载时，rpm将降低，并且负载越大，rpm越小。比如，一个螺旋转头的rpm速度比它在空载时更低。DC电机也有电压和电流的标称值。标称电压是电机在正常情况下运行的电压，可以是1～150 V不等，视电机的实际情况而定。当增加电压时，rpm也会上升。在空载时施加额定电压，此时所消耗的电流就是电流

651

标称值,它可以是从 25 mA 到几安培不等。当负载增加时,rpm 将减少,除非供给电机的电流或电压增加,而这将反过来增大转矩。当电压固定时,随着负载的增大,DC 电机的电流(功率)消耗将会增加。如果使电机超负荷运作,它将会停转,并且会因高电流损耗所产生的热量而遭到损坏。

17.3.2 单方向控制

图 17-13 显示了顺时针方向(CW)和逆时针方向(CCW)DC 电机的旋转。表 17-9 列出了几种 DC 电机。

表 17-9 部分直流电机的特性(www.Jameco.com)

型号	额定电压	电压范围	电流	rpm	转矩
154915CP	3 V	1.5~3 V	0.070 A	5 200	4.0 g-cm
154923CP	3 V	1.5~3 V	0.240 A	16 000	8.3 g-cm
177498CP	4.5 V	3~14 V	0.150 A	10 300	33.3 g-cm
181411CP	5 V	3~14 V	0.470 A	10 000	18.8 g-cm

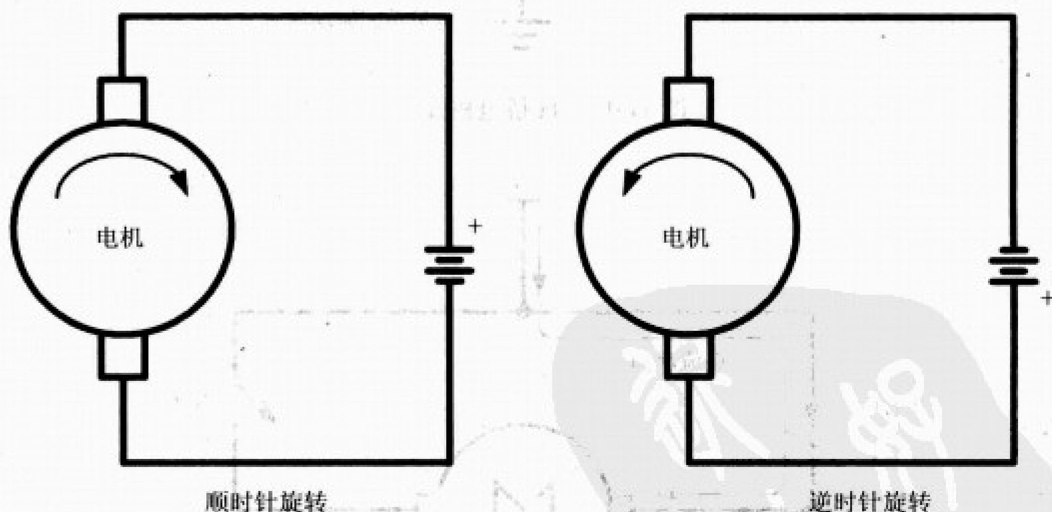


图 17-13 DC 电动机的转动(永久磁场)

17.3.3 双方向控制

通过继电器或其他专门设计的芯片,可以改变 DC 电机旋转的方向。图 17-14~图 17-17 说明了使用 H-桥型电路来控制 DC 电机的基本概念。

图 17-14 说明了如何使用简单的开关来连接 H-桥型电路。这里,所有的开关都处于断开状态,电机不能转动。 652

图 17-15 说明了如何设置开关以使得电机向一个方向转动。当开关 1 和 4 闭合时,电流将流过电机。

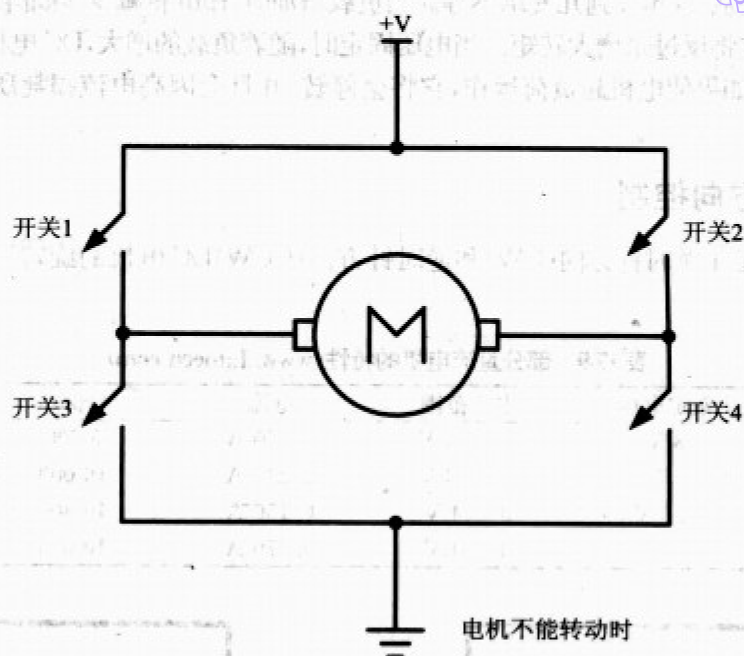


图 17-14　H 桥电机配置

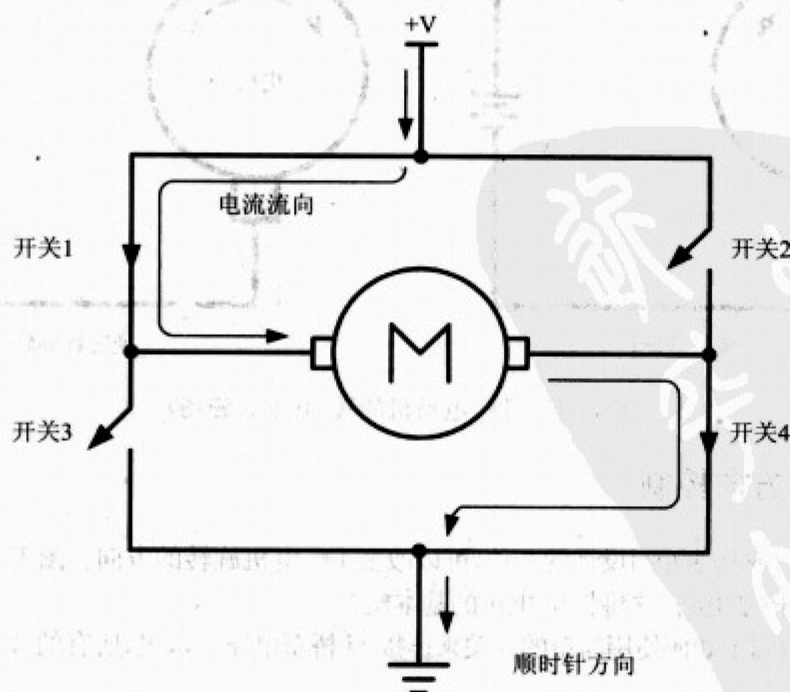


图 17-15　H 桥电机顺时针配置

图 17-16 说明了如何设置开关以使电机向与图 17-15 的相反方向转动。当开关 2 和 3 闭合时，电流将流过电机。

653

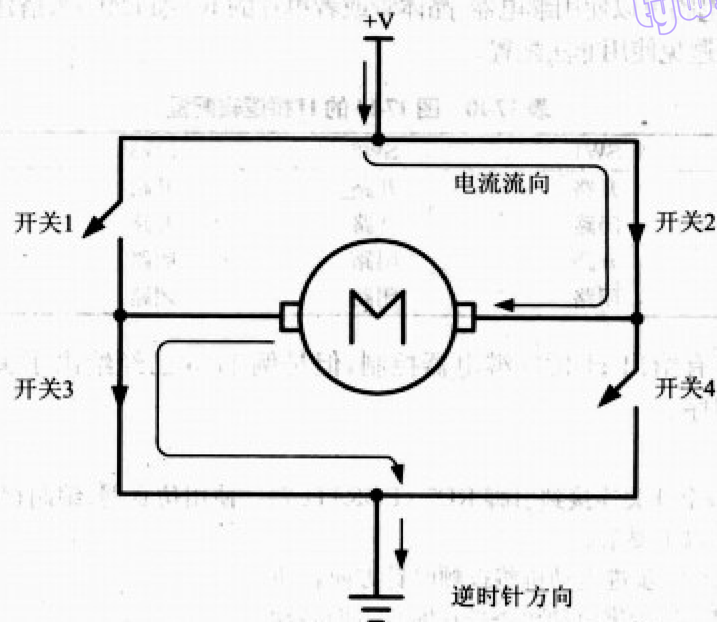


图 17-16 H 桥电机逆时针配置

图 17-17 显示了一个非法配置:将电流直接流向地,造成短路。当开关 1 和 3 闭合,或者开关 2 和 4 闭合时,都将会导致同样的结果。

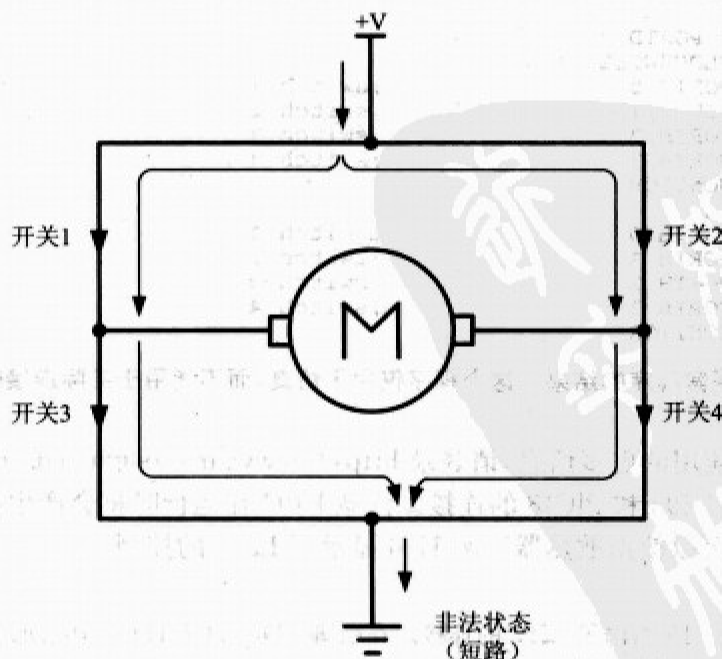


图 17-17 H 桥的非法配置

表 17-10 显示了 H-桥型电路设计中的一些逻辑配置。

H-桥型控制电路可以使用继电器、晶体管或者单片的 IC(如 L293)来搭建。当使用继电器
654
和晶体管时，必须避免使用非法配置。

表 17-10 图 17-14 的 H 桥逻辑配置

电机工作模式	SW1	SW2	SW3	SW4
停止	开路	开路	开路	开路
顺时针旋转	闭路	开路	开路	闭路
逆时针旋转	开路	闭路	闭路	开路
非法	闭路	闭路	闭路	闭路

尽管此处没有给出 H-桥的继电器控制，但是例 17-5 已经给出了关于基本 H-桥操作的一个简单程序。

例 17-5 将一个开关连接到引脚 RD7 (PORTD.7)。使用仿真器，编制程序来仿真表 17-10 中的 H-桥，并完成以下要求。

(a) 如果 DIR=0，步进电动机将按顺时针方向转动。

(b) 如果 DIR=1，步进电动机将按逆时针方向转动。

解：

```
        BCF TRISB,0                ;PORTB.0 as output for switch 1
        BCF TRISB,1                ;      .1  "          switch 2
        BCF TRISB,2                ;      .2  "          switch 3
        BCF TRISB,3                ;      .3  "          switch 4
        BSF TRISD,7                ;make PORTD.7 an input DIR
MONITOR:
        BTFSS  PORTD,7
        BRA CLOCKWISE
        BSF PORTB,0                ;switch 1
        BCF PORTB,1                ;switch 2
        BCF PORTB,2                ;switch 3
        BSF PORTB,3                ;switch 4
        BRA MONITOR
CLOCKWISE:
        BCF PORTB,0                ;switch 1
        BSF PORTB,1                ;switch 2
        BSF PORTB,2                ;switch 3
        BCF PORTB,3                ;switch 4
        BRA MONITOR
```

在仿真器中观察程序的结果。这个程序仅用于仿真，而不能用于实际连接的系统。

关于 H-桥应用的更多信息，请登录 http://www.microdigitaled.com 来获取。图 17-18 给出了 L293 和 PIC18 的连接。注意 L193 在运行时将会产生热量。为了电机的可持续运转，可以使用散热器。例 17-6 显示了 L293 的控制。
655

例 17-6 图 17-18 给出了 L293 的连接。在引脚 RD7 (PORTD.7)处增加了一个开关。编制程序，监视 SW 的状态并完成以下要求。

(a) 如果 SW=0，步进电动机将按顺时针方向转动。

(b) 如果 SW=1，步进电动机将按逆时针方向转动。

解：

```
        BCF   TRISB,0
        BCF   TRISB,1
        BCF   TRISB,2
        BSF   TRISD,7
        BSF   PORTB,0          ;enable the chip
CHK     BTFSS PORTD,7
        BRA   CWISE
        BCF   PORTB,1          ;turn the motor counterclockwise
        BSF   PORTB,2
        BRA   CHK
CWISE   BSF   PORTB,1
        BCF   PORTB,2          ;turn motor clockwise
        BRA   CHK
```

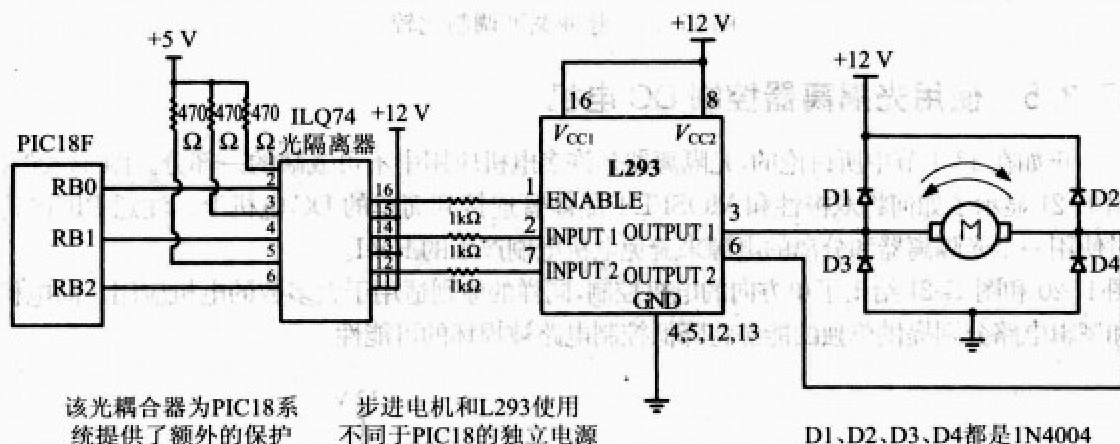


图 17-18　使用 L293 芯片实现双向电机控制

656

17.3.4　脉冲宽度调制(PWM)

电机的转速取决于 3 个因素：负载、电压和电流。对于一个给定的负载，可以通过一种叫作脉冲宽度调制的方法来使电机保持稳定的转速。通过改变施加到 DC 电机上的脉冲宽度，可以增大或减小电机的速度。注意，尽管电压有一个固定的幅度，但它有一个可变的占空比。这意味着脉冲越宽，速度越快。PWM 在 DC 电机控制中得到如此广泛的应用，以至于一些微控制器都将 PWM 电路嵌入到了芯片中。在这样的微处理器中，只需要合理地加载寄存器的值以给出需要的脉冲的高低电平比例，剩下的工作就交给微控制器来完成。这允许微控制器可以做其他的工作。对于没有 PWM 电路的微控制器，可以用软件来产生不同占空比的脉冲，但这将导致微控制器在此期间不能做其他的工作。能够使用 PWM 来控制 DC 电机的转速是 DC 电机比 AC 电机更受欢迎的一个原因。AC 电机的速度由施加到该电机上的电压的 AC频率来确定的，而这个频率通常是固定的。因此当负载增加时，就不能控制 AC 电机的转速。正如先前看到的，还可以改变 DC 电机的方向和转速。图 17-19 给出了不同 PWM 的对比图。

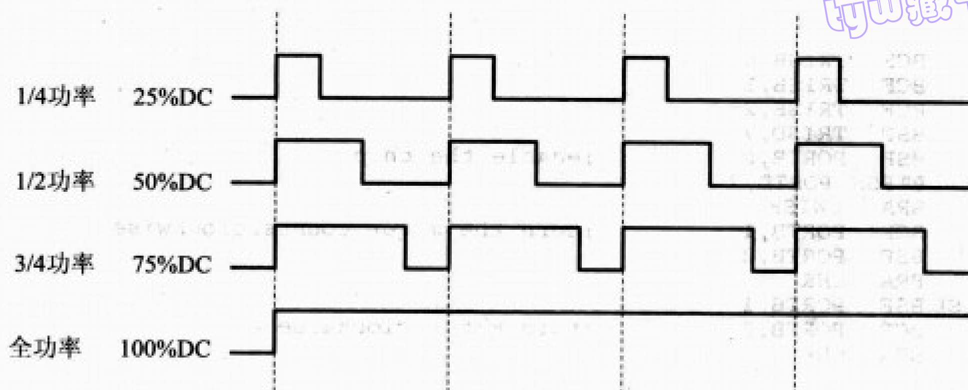


图 17-19 脉冲宽度调制比较

17.3.5 使用光隔离器控制 DC 电机

正如在 17.1 节中所讨论的,光隔离器是许多电机应用中不可或缺的一部分。图 17-20和图 17-21 显示了如何将双极性和 MOSFET 晶体管连接到简单的 DC 电机上。注意 PIC18 通过使用一个光隔离器和分离的电源来避免电机电刷产生的 EMI。

图 17-20 和图 17-21 给出了单方向的电机控制,同样的原则适用于大多数的电机应用。向电机和逻辑电路分别提供单独的能源将降低控制电路被损坏的可能性。

657

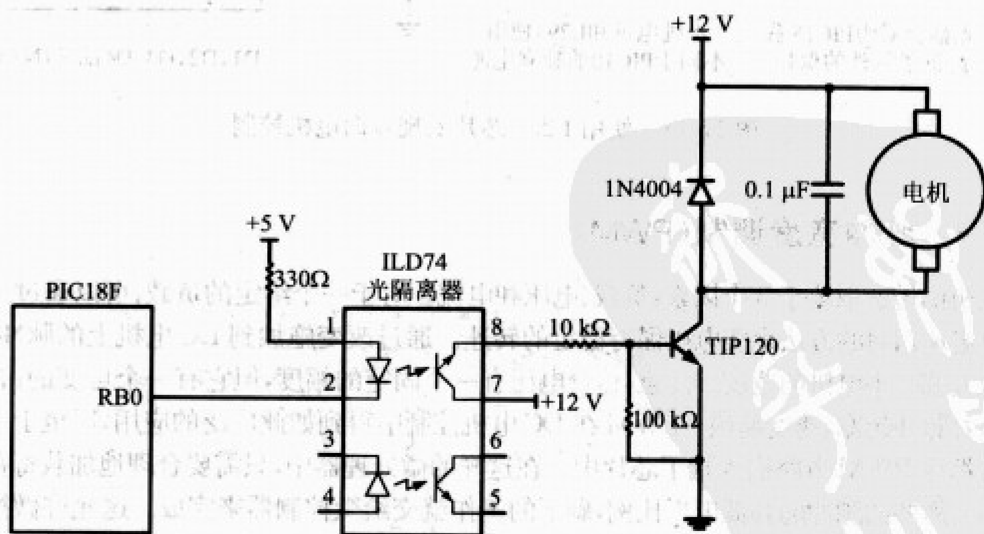


图 17-20 使用达林顿晶体管的 DC 电机连接

图 17-20 显示了双极型晶体管与电机的连接。控制电路的保护由光隔离器提供。电机和 PIC18 分别使用独立的电源供电。相互独立的电源允许使用高电压电机。注意,这里在电机的两端连接了一个去耦电容,这将有助于减少电机产生的 EMI。清零 P1.0 位可以打开电机。

图 17-21 显示了 MOSFET 晶体管的连接。光隔离器保护 PIC18 免受 EMI 的损坏。晶体管

需要用到齐纳二极管，以此来将门限电压降低到额定最大值以下。如例17-7所示。

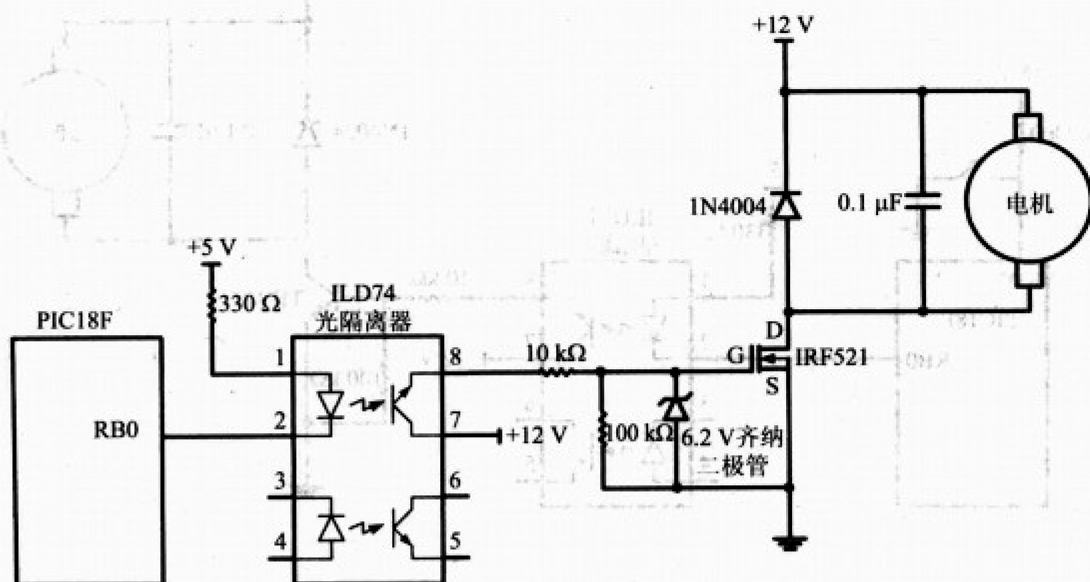


图 17-21 使用MOSFET晶体管的DC电机连接

例 17-7 参考本例中的图。编制程序，监视开关的状态，并完成以下要求。

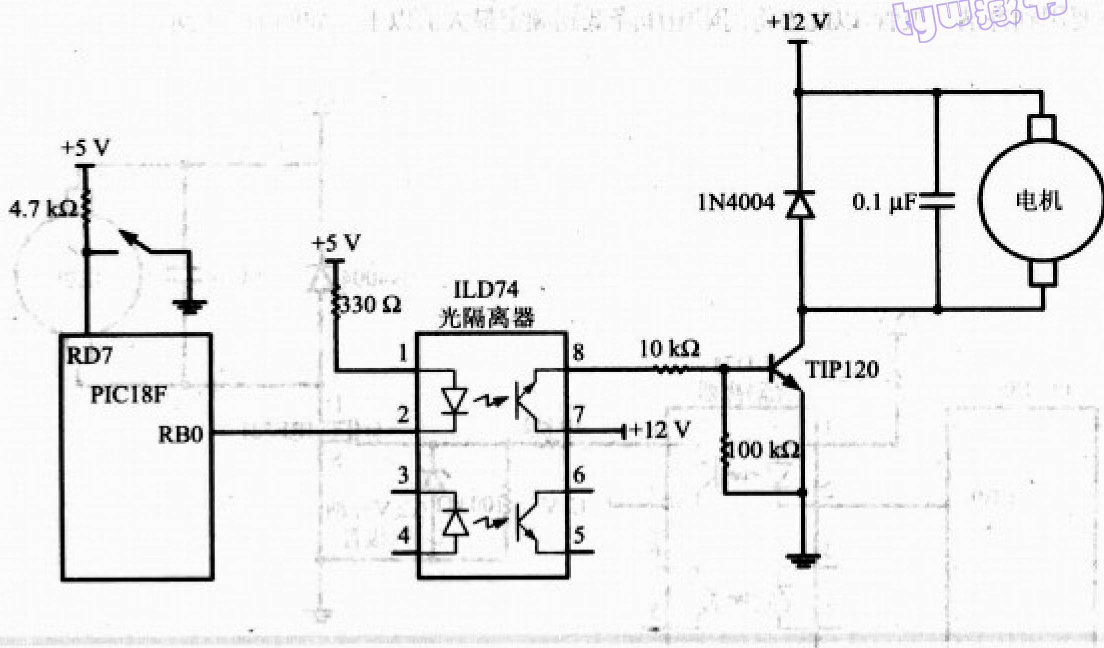
(a) 如果PORTD.7=1，DC电机以占空比为25%的脉冲旋转。

(b) 如果PORTD.7=0，DC电机以占空比为50%的脉冲旋转。

解：

```
        BCF TRISB,RB0          ;PORTB.0 as output
        BSF TRISD,RD7          ;PORTD.7 as input
        BCF PORTB,RB0          ;turn off motor
CHK
        BTFSS PORTD,RD7
        BRA   PWM_50
        BSF PORTB,RB0          ;high portion of pulse
        CALL DELAY
        BCF PORTB,RB0          ;low portion of pulse
        CALL DELAY
        CALL DELAY
        CALL DELAY
        BRA   CHK
PWM_50
        BSF PORTB,RB0          ;high portion of pulse
        CALL DELAY
        CALL DELAY

        BCF PORTB,RB0          ;low portion of pulse
        CALL DELAY
        CALL DELAY
        BRA CHK
```



659

17.3.6　DC 电机的控制和 PWM 的 C 编程

例 17-8～例 17-10 将给出对应于前面的 PIC18 控制 DC 电机程序的 C 语言版本。

例 17-8　参考图 17-18 的电机连接。将一个开关连接到引脚 RD7。编制 C 程序，监视 SW 的状态并完成以下要求。

(a) 如果 SW=0，步进电动机将按顺时针方向转动。

(b) 如果 SW=1，步进电动机将按逆时针方向转动。

解：

```
#include <p18f458.h>

#define SW PORTDbits.RD7
#define ENABLE PORTBbits.RB0
#define MTR_1 PORTBbits.RB1
#define MTR_2 PORTBbits.RB2

void main()
  {
  TRISD=0x80;    //make RD7 input pin
  TRISB=0x0;     //make PORTB output
  SW = 1;
  ENABLE = 0;
  MTR_1 = 0;
  MTR_2 = 0;

  while(1)
    {
```

```
     ENABLE = 1;
     if(SW == 1)
       {
         MTR_1 = 1;
         MTR_2 = 0;
       }
     else
       {
         MTR_1 = 0;
         MTR_2 = 1;
       }
   }
}
```

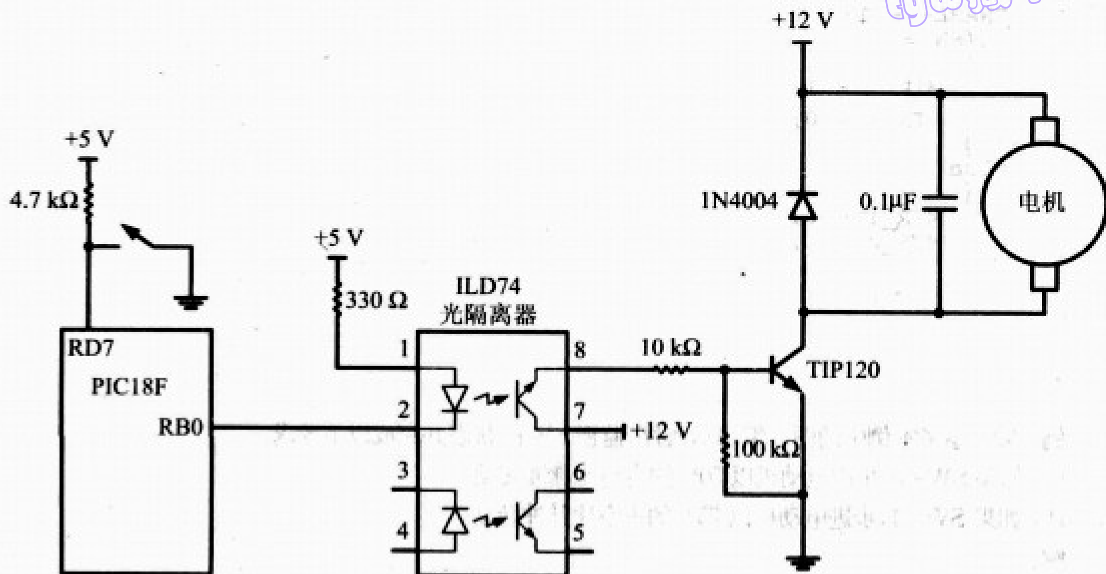

例 17-9 参考本例中的图。编写 C 程序，监视 SW 的状态并完成以下要求。

(a) 如果 SW=0，步进电动机以 50%的占空比脉冲转动。

(b) 如果 SW=1，步进电动机以 25%的占空比脉冲转动。

解：

```
#include <p18f458.h>
#define SW PORTDbits.RD7
#define MTR  PORTBbits.RB1
void MSDelay(unsigned int value);
void main()
  {
    TRISD=0x80;         //make RD7 input pin
    TRISB=0xFD;         //make RB1 output pin
    while(1)
      {
        if(SW == 1)
          {
            MTR = 1;
            MSDelay(25);
            MTR = 0;
            MSDelay(75);
          }
        else
          {
            MTR = 1;
            MSDelay(50);
            MTR = 0;
            MSDelay(50);
          }
      }
  }
void MSDelay(unsigned int value)
  {
    unsigned char x, y;
    for(x=0; x<1275; x++)
      for(y=0; y<value; y++);
  }
```



661

例 17-10　参见图 17-20 电动机的连接。引脚 RD0 和 RD1 均连接了两个开关。编写 C 程序，监视两个开关的状态并完成以下要求。

SW2(RD1)	SW1 (RD0)	
0	0	DC 电机慢速转动(25%占空比)
0	1	DC 电机中速转动(50%占空比)
1	0	DC 电机快速转动(75%占空比)
1	1	DC 电机极快速转动(100%占空比)

解：

```
#include <p18f458.h>
#define MTR PORTBbits.RB1
void MSDelay(unsigned int value);

void main()
  {
  unsigned int duty;
  TRISB = 0xFD;
  TRISD = 0xFF;
  while(1)
     {
      duty = PORTD&0x03;
      duty++;
      duty *= 25;
      MTR = 1;
      MSDelay(duty);
      MTR = 0;
      MSDelay(100-duty);
     }
  }
```

17.3.7　复习题

1. 判断对错：永久性磁场的 DC 电机只有两根引线用于正和负电压。

2. 判断对错:同步进电机的控制一样,也可以精确控制 DC 电机转动的角度。
3. 为什么要在微控制器和 DC 电机之间放置一个驱动器?
4. 如何改变 DC 电机的转动方向?
5. DC 电机中停转是指什么?
6. 判断对错:PWM 允许使用相同的相位但不同幅度的脉冲来控制 DC 电机。
7. DC 电机给定的 RPM 标称值是指________(空载,有载)。

662

17.4 使用 CCP 来控制 PWM 电机

在第 15 章已经讨论了 PIC452/458 的 CCP(比较捕获脉冲宽度调制)。CCP 的特征之一就是脉冲宽度调制(PWM),正如在 15.4 节中看到的。在这节中将使用 CCP 的 PWM 特征来控制 DC 电机。在开始学习本节之前,请温习 15.4 节中的 PWM 编程。

17.4.1 使用 CCP 来控制 DC 电机

回忆 15.4 节所述内容,CCP 的 PWM 部分是通过 PR2 和定时器 2 寄存器来编程的。程序 17-2 使用了 CCP1 的 PWM 特征来重写的例 17-7。注意程序 17-2 是第 15 章中的程序 15-5 的修订版本。程序 17-2C 是程序 17-2 的 C 语言版本。在程序 17-2 中,不断地监视输入开关。如果开关是低电平,PIC18 将使用 CCP1 模型产生一个占空比为 50%的 PWM。如果开关为高电平,PIC18 将产生一个占空比为 25%的 PWM。回忆第 15 章所述内容,必须使用 PR2 和定时器 2 寄存器产生 PWM 脉冲。

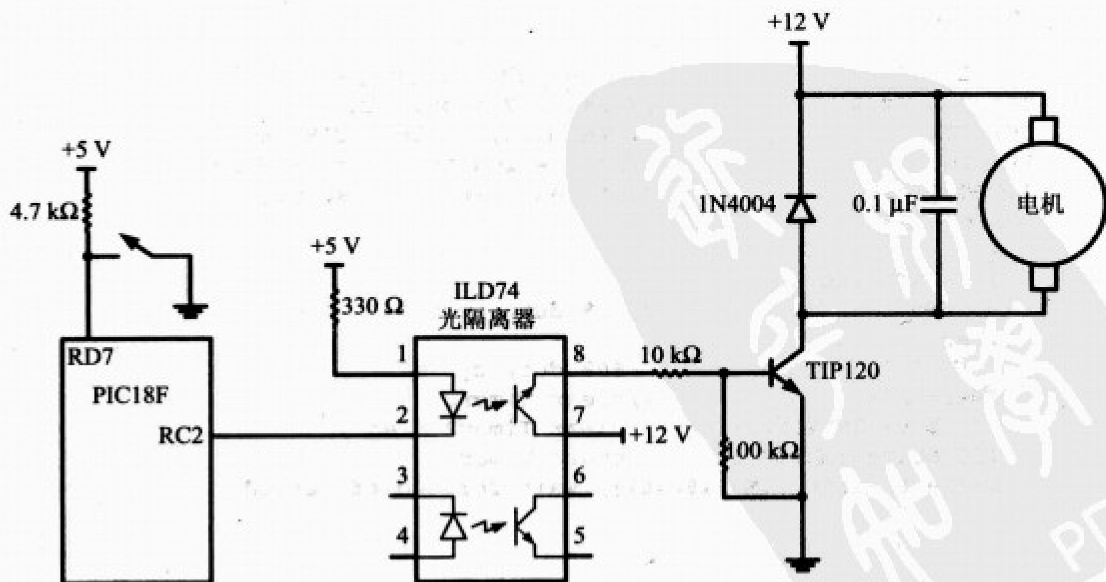


图 17-22 使用 CCP1 引脚的 DC 电机控制

程序 17-2

```
;Program 17-2
       BCF   TRISC,CCP1        ;make PWM output pin
       BSF   TRISD,RD7         ;make RD7 input pin
       MOVLW 0x3C              ;PWM MODE, 11 for DC1B1:B0
       MOVWF CCP1CON
       MOVLW D'100'            ;set period to 100 * Fosc/4
       MOVWF PR2
       MOVLW 0x01        ;Timer2, 4 prescale, no postscaler
       MOVWF T2CON
AGAIN  BTFSS PORTD,RD7         ;Is the switch high?
       BRA T2DUTY              ;no, then 50%
       MOVLW D'25'             ;25% duty cycle
       BRA LOAD
T2DUTY MOVLW D'50'             ;50% duty cycle
       BRA LOAD
LOAD   MOVWF CCPR1L            ;load duty cycle
       CLRF   TMR2             ;clear Timer2
       BSF    T2CON,TMR2ON     ;turn on Timer2
       BCF    PIR1,TMR2IF      ;clear Timer2 flag
OVER   BTFSS PIR1,TMR2IF       ;wait for end of period
       BRA    OVER
       GOTO   AGAIN            ;continue
```

663

以下是上述程序的C语言版本。

程序 17-2C

```
//Program 17-2C
#include <p18f458.h>
void main()
  {
  TRISC = 0xFB;                //make CCP1 output pin
  TRISD = 0x80;                //make RD7 input pin
  CCP1CON = 0x3C;              //PWM MODE, 11 for DC1B1:B0
  PR2=100;                     //set period to 100 * 16/Fosc
  T2CON=0x01;                  //4 prescaler, no postscaler
  while(1)
    {
    if(PORTDbits.RD7==1)
      CCPR1L = 25;             //25% duty cycle
    else
      CCPR1L = 50;             //50% duty cycle
    TMR2=0x0;                  //clear Timer2
    PIR1bits.TMR2IF=0;         //clear Timer2 flag
    T2CONbits.TMR2ON=1;        //start Timer2
    while(PIR1bits.TMR2IF==0);//wait for end of period
    }
  }
```

17.4.2　复习题

1. 判断对错：对于标准的CCP1，将RC2引脚用于PWM。
2. 判断对错：对于标准的CCP1，CCP1引脚必须设置为输出。
3. 在标准的CCP1中，使用________来设置PWM的周期。

4. 在标准的 CCP1 中，使用________来设置 PWM 的负载周期。
5. 判断对错：在标准的 CCP1 中，必须将定时器 1 用于 PWM。

664

17.5 使用 ECCP 来控制 DC 电机

PIC18F452/458(或 450/4580)有一个标准的 CCP 和一个增强的 CCP(ECCP)。实际上，近年来 CCP 模型的被重视程度下降了，而 ECCP 变成了 PIC18 家族的新宠。其原因就在于，除了支持标准 CCP 中的捕获/比较模式外，ECCP 还允许 H-桥来实现对 DC 电机的双向控制。在这一节中，将使用 PIC18 的 ECCP 特征来控制 DC 电机。在开始学习本节之前，请复习第 15 章中关于 ECCP 编程的基本概念。

17.5.1 使用 ECCP 来双向控制 DC 电机

由于使用的是 4 个引脚而不是标准 CCP 中的单引脚，ECCP 允许控制 DC 电机双向转动的 H-桥实现。正如在 17.3 节中所看到的，双向的 DC 转动需要一些 H-桥电路。PIC18 的 ECCP 模型在内部实现了完整的 H-桥电路。通过 RD7～RD4(PORTD.7～PORTD.4)可以实现对直流电机的双向控制，如图 17-23～图 17-26 所示。

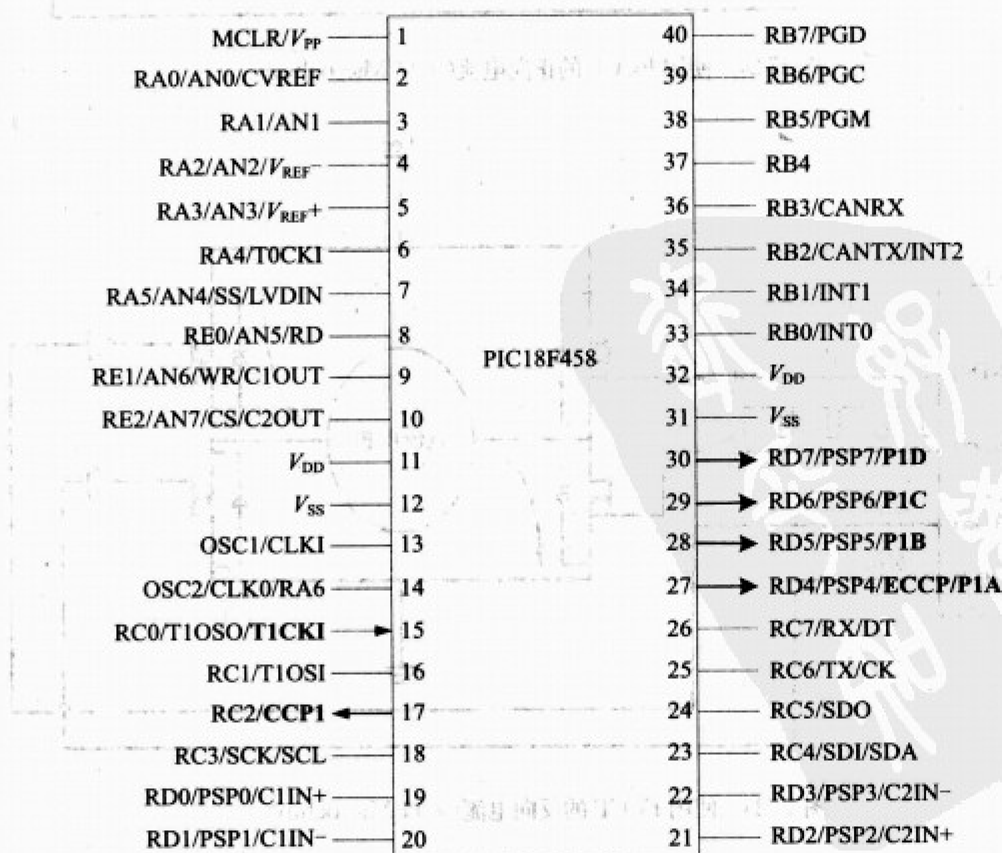


图 17-23 PIC18F458/4580(452/4520)中用于 PWM 的 ECCP 引脚

665

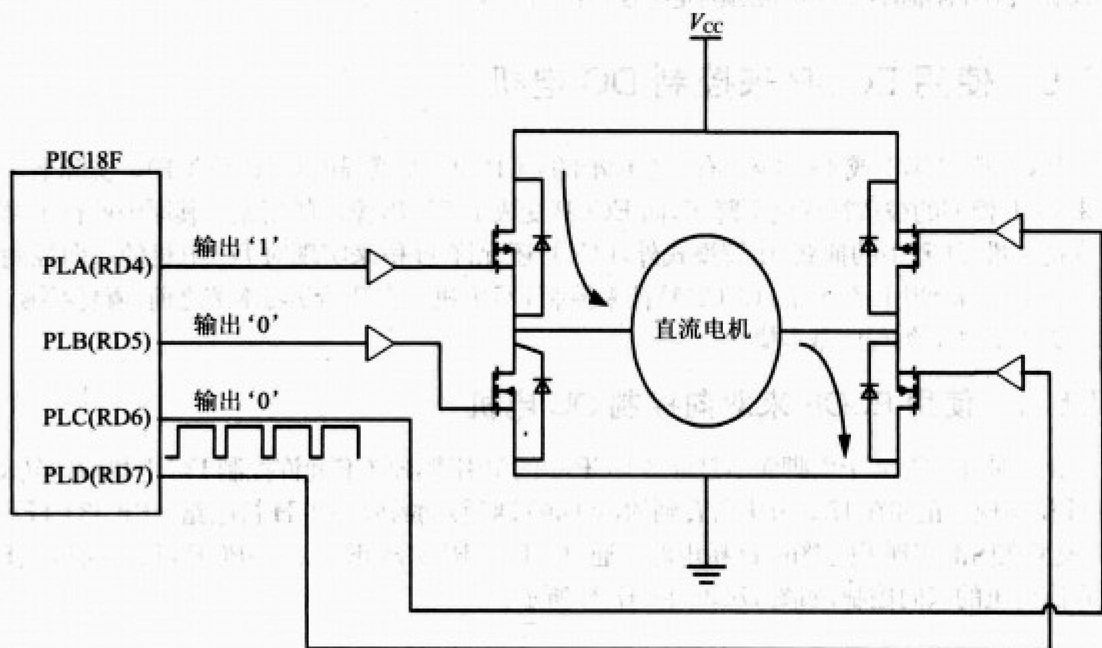


图 17-24　使用 ECCP 的正向电流(来自 Microchip)

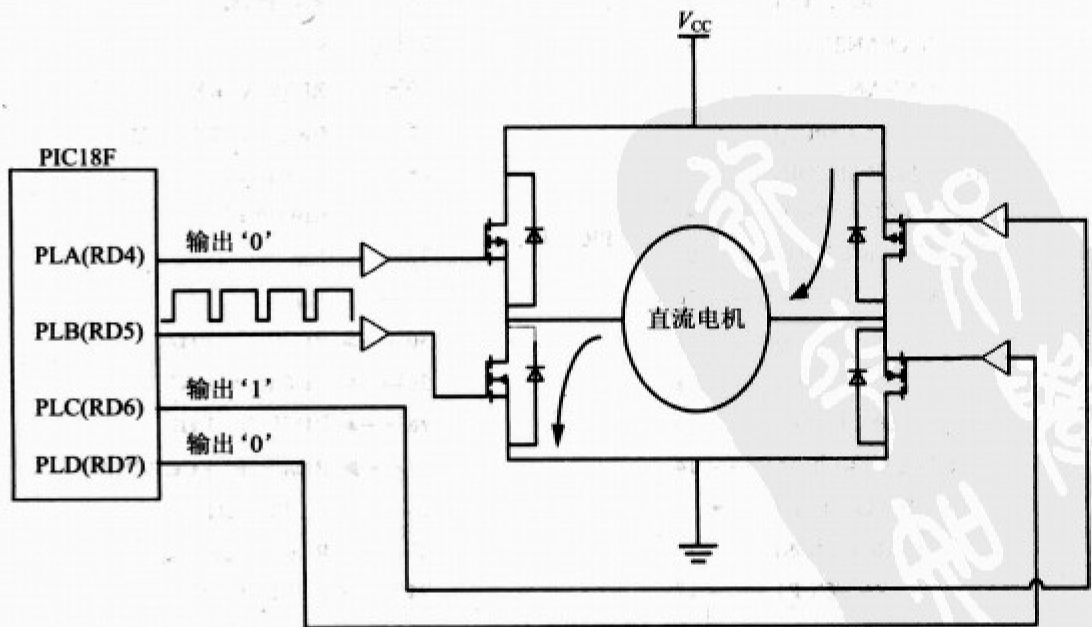


图 17-25　使用 ECCP 的反向电流(来自 Microchip)

EPWM1M1	EPWM1M0	EDC1B1	EDC1B0	ECCP1M3	ECCP1M2	ECCP1M1	ECCP1M0
D7							D0

EPWM1M1：EPWM1M0 PWM 输出引脚设置。它允许在捕获/比较模式下使用单引脚，或者在 PWM 下使用 4 个引脚

在捕获/比较模式下只使用了引脚。在这种情况下，无需对这两位做出选择

在 PWM 模式下，这两位的选项如下

00 P1A 被用作调制输出。P1B、P1C 和 P1D 被用作 I/O 引脚

01 全桥前向输出，P1D 调制，P1A 激活。P1B、P1C 未激活

10 半桥输出 。P1A 和 P1D 调制，带死区控制，P1C 和 P1D 被用作 I/O 引脚

11 全桥反向输出。P1B 调制，P1C 激活，P1A 和 P1D 未激活

EDC1B 10：EDC1B1 PWM 占空比的最低有效位。只使用在 PWM 下。10 位的占空比寄存器的最小有效位(第 1 位和第 0 位)用于 PWM，而 ECCPR1L 寄存器被用作 10 位占空比寄存器的第 2 位到第 9 位

ECCP1 M3～ECC1M0 ECCP 模式选择

0000 ECCP1 关闭

0001 保留

0010 比较模式。在匹配时翻转 ECCP1 的输出引脚(设置 ECCP1IF 位)

0011 保留

0100 捕获模式，每个下降沿捕获

0101 捕获模式，每个上升沿捕获

0110 捕获模式，每第 4 个上升沿捕获

0111 捕获模式，每第 16 个上升沿捕获

1000 比较模式。初始化 ECCP1 引脚为低电平；在匹配时，使 CCP1 引脚为高电平

1001 比较模式。初始化 ECCP1 引脚为高电平；在匹配时，使 CCP1 引脚为低电平

1010 比较模式。在匹配时产生软件中断

1011 比较模式。触发特殊事件

1100 PWM 模式。P1A 和 P1C 高电平有效，P1B 和 P1D 高电平有效

1101 PWM 模式。P1A 和 P1C 高电平有效，P1B 和 P1D 低电平有效

1110 PWM 模式。P1A 和 P1C 低电平有效，P1B 和 P1D 高电平有效

1111 PWM 模式。P1A 和 P1C 低电平有效，P1B 和 P1D 低电平有效

图 17-26 ECCP1 控制寄存器(该寄存器用于选择捕获、比较或者 EECP1 的 PWM 3 种操作模式之一)

程序 17-3 给出了基于 ECCP 模块的 PWM 的全桥实现。关于基于 ECCP 模块的半桥实现和其他的应用，请参考 PIC18 的手册。

程序 17-3

```
;Program 17-3
      CLRF TRISD            ;make PORTD output
      MOVLW D'100'
      MOVWF PR2             ;period = 100 * 16/Fosc
      MOVLW D'50'
      MOVWF ECCPR1L         ;duty = 50%
      MOVLW 0xCF
      MOVWF ECCP1CON        ;reverse full-bridge PWM
      MOVLW 0x24
      MOVWF T2CON           ;4 postscaler, turn on Timer2
AGAIN CLRF  TMR2            ;start pulse
      BCF   PIR1,TMR2IF     ;clear flag
WAIT  BTFSS PIR1,TMR2IF     ;wait for period
      BRA   WAIT
      BRA   AGAIN           ;do it again
```

以下是上述程序的 C 语言版本。

程序 17-3C

```
//Program 17-3C
#include <p18f458.h>

void main()
  {
     TRISD=0;                 //make PORTD output
     PR2=100;                 //period = 100 * 16/Fosc
     ECCPR1L=50;              //duty = 50%
     ECCP1CON=0xCF;           //reverse full-bridge PWM
     T2CON=0x24;              //4 postscaler,turn on Timer2
while(1)
     {
     TMR2=0;                  //start pulse
     PIR1bits.TMR2IF=0;            //clear flag
     while(PIR1bits.TMR2IF==0);  //wait for period
     }
  }
```

17.5.2 复习题

1. 判断对错:对于 ECCP1,将 RD3~RD0 引脚用于全桥电路。
2. 判断对错:对于 ECCP1,P1A~P1D 引脚必须设置为输出。
3. 在 ECCP1 中,使用________来设定 PWM 周期。
4. 在标准的 ECCP1 中,使用________来设置 PWM 的占空比。
5. 判断对错:在标准的 ECCP1 中,必须将定时器 2 用于 PWM。

小结

这一章继续讲述了如何将 PIC18 和实际生活中的设备相连接。这章讨论的设备有继电器、光隔离器、步进电机和 DC 电机。

首先,定义了继电器和光隔离器的基本操作,以及用于描述和控制其操作的关键术语。

接着讲解了 PIC18 和步进电机的连接。然后使用 PIC18 汇编和 C 语言编程，以实现通过光隔离器来控制步进电机的目的。

PIC18 还可以与 DC 电机相连。一个典型的 DC 电机可以将电脉冲转化为机械运动。这章展示了 PIC18 和 DC 电机的接口。接着给出了一些简单的汇编和 C 语言程序来阐释 PWM 的概念。

需要使用电机的控制系统必须针对所采用的电机类型来评估。比如说，不希望在高速设备中使用步进电机，或者在低速、高转矩的设备中使用 DC 电机。步进电机是开环位置系统的理想选择，而 DC 电机将在高速传送带应用中发挥更好的作用。DC 电机可以通过引入传动轴编码器来实现闭环控制系统，接着使用微控制器来监视电机的精确位置和速度。在最后两节介绍了如何使用 PIC18 的 CCP 和 ECCP 特征来控制 DC 电机。

习题

1. 判断对错：所有继电器的最小磁化电压都是一样的。
2. 判断对错：磁化继电器需要的最小电流由线圈电阻决定。
3. 请指出固态继电器优于 EM 继电器的地方。
4. 判断对错：在继电器中，磁化电压和触点电压是一样的。
5. 设线圈电阻为 1200 Ω，线圈电压为 5 V。计算继电器的磁化电流。
6. 请指出光隔离器的两种应用。
7. 请指出光隔离器相对于 EM 继电器的优势。
8. 对于 EM 继电器和固态继电器，哪一个会有反向 EMF 问题？
9. 判断对错：线圈电阻越大，则反向 EMF 电压越大。
10. 判断对错：对于线圈电压和触片电压，必须使用相同的电压源。
11. 如果一个电机需要 90 步才能完成一圈完整的旋转，那么它的步进角度是多少？
12. 如果步进角是 7.5°，求每转一圈所需要的步数？
13. 如果第一步是 0011(二进制)，请给出顺时针方向的正常四步顺序。
14. 如果第一步是 1100(二进制)，请给出顺时针方向的正常四步顺序。
15. 如果第一步是 1001(二进制)，请给出逆时针方向的正常四步顺序。
16. 如果第一步是 0110(二进制)，请给出逆时针方向的正常四步顺序。
17. 在 PIC18 和步进电机之间放置 ULN2003 的作用是什么？能够用于 3 A 的电机吗？
18. 以下哪个不能作为步进电机的正常四步顺序：
    (a) CCH (b) DDH (c) 99H (d) 33H
19. 在每步之间插入时延的作用是什么？
20. 在第 19 题中，如何使步进电机转动得更快？
21. 哪种电机最适合精确地转动 90°？
22. 判断对错：DC 电机的电流消耗与负载成正比。
23. 判断对错：在空载和有负载的情况下，DC 电机的 rpm 是一样的。
24. 在数据表中的 rpm 是用于________(空载，有负载)的。
25. DC 电机相比 AC 电机的优势在哪里？
26. 步进电机相比 DC 电机的优势在哪里？

27. 判断对错:如果供给电机的电压和电流是恒定的,那么在DC电机上施加更大的负载,电机的速度将会慢下来。
28. 什么是PWM? 怎样使用它来控制DC电机?
29. 一个直流电机正在移动一个负载。如何将rpm保持恒定呢?
30. 在电机和微控制器之间放置光隔离器的作用是什么?

复习题答案

17.1 节

1. 对于继电器,可以使用5 V的数字系统来控制12~120 V的设备(如喇叭和其他仪器)。
2. 因为微控制器/数字输出没有足够的电流来磁化继电器,所以需要一个驱动器。
3. 当线圈未被磁化时,触片是闭合的。
4. 当电流流过线圈时,线圈周围将产生磁场,这个磁场将使铁芯被线圈吸引。
5. 响应速度更快,而且磁化所需的电流更少。
6. 体积更小,而且可以不使用驱动器就直接连接到微控制器上。

17.2 节

1. 对于顺时针为0110 0011 1001 1100;对于逆时针为0110 1100 1001 0011。
2. 72

670

3. 因为微控制器的引脚不能提供足够的电流来驱动步进电机。

17.3 节

1. 正确。 2. 错误。
3. 因为微控制器/数字输出缺少足够的电流来磁化继电器,所以需要一个驱动器。
4. 通过将连接到引线上的电压的极性反向。
5. 如果所连接的负载超过其承受能力,DC电机将停转。
6. 错误。 7. 空载。

17.4 节

1. 正确。 2. 正确。 3. PR2。 4. CCPR1L。 5. 错误。

17.5 节

671

1. 错误。 2. 正确。 3. PR2。 4. CCPR1L。 5. 正确。

附录 A PIC18 指令:格式和描述

概述

A.1 节将介绍 PIC18 的指令格式,其中特别介绍了使用 WREG 寄存器和文件寄存器的指令。在介绍指令格式的同时,还给出了每一条 PIC18 指令的机器周期。

A.2 节将具体地介绍每一条 PIC18 指令。在大多数情况下,都使用一个简单的程序来说明指令的用法。

本附录主要介绍 PIC18 的指令。A.1 节将介绍指令的格式与种类;A.2 节使用程序例子来阐述每一条指令的用法。

A.1 PIC18 指令格式与类别

如图 A-1 所示,PIC18 的指令分为 5 大类:

(1) 面向位的指令

(2) 使用立即数的指令

(3) 面向字节的指令

(4) 表读写指令

(5) 使用分支和调用的控制指令

本节将介绍指令的格式和语法,尤其是面向字节的指令。对于其中的一些指令,读者需要回到第 6 章(6.3 节)去回顾访问存储区和存储区寄存器的有关概念。

1. 面向位的指令

面向位的指令是对文件寄存器的指定位进行操作的。在操作完成后,结果放回到同一个文件寄存器中。例如,指令"BCF f,b,a"用来对文件寄存器的指定位清零,如表 A-1 所示。在这类指令中,b 代表文件寄存器里的指定位,可以取值为 0~7,表示寄存器的 D0~D7 位。文件寄存器的位置可以是存储区寄存器[即访问存储区(若 a=0)],也可以是其他的存储区寄存器(若 a=1)。注意,如果 a=1,编译器会自动地默认是访问存储区。

表 A-1 面向位的指令（来自 Microchip 的数据表）

伪指令，操作数		描 述	周 期
面向位的文件寄存器操作			
BCF	f,b,a	将位清零	1
BSF	f,b,a	将位置 1	1
BTFSC	f,b,a	位测试，若为 0 则跳转	1(2 或 3)
BTFSS	f,b,a	位测试，若为 1 则跳转	1(2 或 3)
BTG	f,d,a	位翻转	1

674

面向字节的文件寄存器操作　　指令例子

15–10: 操作码 | 9: d | 8: a | 7–0: 文件寄存器　　ADDWF MYREG,W,B

d=0，目标地址是WREG寄存器
d=1，目标地址是文件寄存器
a=0，强制制定为访问存储区
a=1，由BSR来选择存储区
f=8位文件寄存器地址

从字节到字节的传送操作（双字）

15–12: 操作码 | 11–0: 源文件寄存器地址　　MOVFF MYREG1,MYREG2

15–12: 1111 | 11–0: 目标文件寄存器地址

12位的文件寄存器地址

面向位的文件寄存器操作

15–12: 操作码 | 11–9: b(BIT #) | 8: a | 7–0: 文件寄存器　　BSF MYREG,bit,B

b=文件寄存器的3位地址
a=0，强制指定为访问存储区
a=1，由BSR来选择存储区
f=8位的文件寄存器地址

立即数操作

15–8: 操作码 | 7–0: k（立即数）　　MOVLW 0×7F

k=8位的立即数

控制操作

调用，跳转和分支操作

15–8: 操作码 | 7–0: n<7:0>（立即数）　　GOTO 标签

15–12: 1111 | 11–0: n<19:8>（立即数）

n=20位的立即数

675

图 A-1 PIC18 指令的通用格式（来自 Microchip）

为更清晰地区分面向位的指令，请看下面的若干例子：

```
BCF   PORTB,5          ;clear bit D5 of PORTB
BCF   TRISB,4          ;clear bit D4 of TRISC reg
BTG   PORTC,7          ;toggle bit D7 of PORTC
BTG   PORTD,0          ;toggle bit D0 of PORTD
BSF   STATUS,C         ;set carry flag to one
```

下面是使用访问存储区中文件寄存器的例子：

```
MyReg    SET   0x30    ;set aside loc 30H for MyReg
MOVLW 0x0              ;WREG = 0
MOVWF MyReg            ;MyReg = 0
BTG MYReg,7            ;toggle bit D7 of MyReg
BTG MYReg,5            ;toggle bit D5 of MyReg
```

下面还是使用访问寄存区中文件寄存器的例子：

```
MyReg    SET    0x50 ;set aside loc 50H for MyReg
MOVLW    0x0         :WREG = 0
MOVWF    MyReg       ;MyReg = 0
BTG     MYReg,2      ;toggle bit D2 of MyReg
BTG     MYReg,4      ;toggle bit D4 of MyReg
```

正如在第 6 章介绍过的，当使用访问存储区以外的存储区时，必须由 BSR(存储区选择寄存器)来指定目标存储区的编号，编号可以是从 0 到 F(十六进制)的任意值。这可以使用 MOVLB 指令来实现。请看下面的例子。

下面的例子使用了存储区 2 的地址(RAM 地址范围为 200～2FFH)。

```
YReg   SET 0x30        ;set aside loc 30H for YReg
MOVLB 0x2         ;use Bank 2 (address loc 230H)
MOVLW 0x0              :WREG = 0
MOVWF YReg             ;YReg = 0
BTG   YReg,7,1    ;toggle bit D7 of YReg in bank 2
BTG   YReg,5,1    ;toggle bit D5 of YReg in bank 2
```

下面的例子使用了存储区 4 的地址(RAM 地址范围为 400～4FFH)。

```
ZReg   SET 0x10        ;set aside loc 10H for ZReg
MOVLB 0x4         ;use Bank 4 (address loc 410H)
MOVWL 0x0              ;WREG = 0
MOVWF ZReg             ;ZReg = 0
BSF ZReg,6,1 ;set HIGH bit D6 of ZReg in bank 4
BSF ZReg,1,1 ;set HIGH bit D1 of ZReg in bank 4
```

注意，所有面向位的指令都是以字母 B(位)开始的。分支指令也是以字母 B 开始，如 BZ target，若为零则跳转，但它们不是面向位的指令。

676

2. 使用立即数的指令

这类指令对 WREG 寄存器的内容和固定的数值 k 进行操作，如表 A-2 所示。因为 WREG 寄存器只有 8 位，所以 k 不能高于 8 位。因此，k 的值是 0～255(十六进制为 00～FFH)。在操作完成后，结果将被放回 WREG 中。请看下面的例子：

```
MOVLW    0x45 ;WREG = 45H
ADDLW    0x24 ;WREG = 45H + 24H = 69H

MOVLW    0x35 ;WREG = 35H
ANDLW    0x0F ;WREG = 35H ANDed with 0FH = 05H

MOVLW    0x55 ;WREG = 55H
XORLW    0xAA ;WREG = 55H EX-ORed with AAH = FFH
```

表 A-2 立即数指令(来自 Microchip 的数据表)

伪指令,操作数		描 述	周 期
		立即数操作	
ADDLW	k	将立即数和 WREG 相加	1
ANDLW	k	将立即数和 WREG 做与运算	1
IORLW	k	将立即数和 WREG 做逻辑或运算	1
LFSR	f,k	把立即数(12 位)第二个字传送给 FSRx	2
MOVLB	k	将立即数传送入 BSR<3:0>	1
MOVLW	k	将立即数送入 WREG 寄存器	1
MULLW	k	将立即数和 WREG 相乘	1
RETLW	k	在 WREG 中带立即数的返回	2
SUBLW	k	将立即数减去 WREG 寄存器的内容	1
XORLW	k	将立即数和 WREG 的内容做逻辑异或运算	1

3. 面向字节的指令

这类指令又可以分成两组。第一组是,指令对文件寄存器进行操作,结果放回到文件寄存器。例如"CLRF f, a"指令就是其中之一。如表 A-3 所示。第二组是,指令同时涉及文件寄存器和 WREG 寄存器。因此,可以选择将结果存放在文件寄存器还是 WREG 寄存器。如指令"ADDWF f, d, a"就是其中一条。存放结果的目标地址可以是 WREG 寄存器(若 d=0),也可以是文件寄存器(若 d=1)。对于文件寄存器的地址,可以是访问存储区(若 a=0),也可以是其他的存储区(若 a=1)。同样要注意的是,当 a=1 时,编译器会自动地选择默认值。

677

表 A-3 面向字节的指令(来自 Microchip 的数据表)

伪指令,操作数		描 述	周 期
		面向字节的文件寄存器操作	
ADDWF	f,d,a	将 WREG 寄存器的内容和文件寄存器的内容相加	1
ADDWFC	f,d,a	将 WREG 的内容和文件寄存器的内容与进位相加	1
ANDWF	f,d,a	将 WREG 的内容和文件寄存器的内容做逻辑与运算	1
CLRF	f,a	将文件寄存器的内容清零	1
COMF	f,d,a	将文件寄存器的内容做取补运算	1
CPFSEQ	f,a	比较文件寄存器的内容和 WREG 寄存器的内容,若相等则跳转	1
CPFSGT	f,a	比较文件寄存器的内容和 WREG 寄存器的内容,若大于则跳转	1

（续）

伪指令，操作数		描　述	周　期
		面向字节的文件寄存器操作	
CPFSLT	f,a	比较文件寄存器的内容和 WREG 寄存器的内容，若小于则跳转	1
DECF	f,d,a	将文件寄存器的内容减 1	1
DECFSZ	f,d,a	将文件寄存器的内容减 1，若为 0 则跳转	1
DCFSNZ	f,d,a	将文件寄存器的内容减 1，若非 0 则跳转	1
INCF	f,d,a	将文件寄存器的内容加 1	1
INCFSZ	f,d,a	将文件寄存器的内容加 1，若为 0 则跳转	1
INFSNZ	f,d,a	将文件寄存器的内容加 1，若非 0 则跳转	1
IORWF	f,d,a	将文件寄存器的内容和 WREG 寄存器的内容做逻辑或运算	1
MOVF	f,d,a	传送文件寄存器的内容	1
MOVFF	f_s,f_d	将 f_s(源)寄存器的内容传送给第一个字，将 f_d(目的)寄存器的内容传送给第二个字	2
MOVWF	f,a	把 WREG 的内容复制到文件寄存器	1
MULWF	f,a	将 WREG 和文件寄存器相乘	1
NEGF	f,a	将文件寄存器取反	1
RLCF	f,d,a	带进位/借位循环左移	1
RLNCF	f,d,a	不带进位/借位循环左移	1
RRCF	f,d,a	带进位/借位循环右移	1
RRNCF	f,d,a	不带进位/借位循环右移	1
SETF	f,a	将文件寄存器的内容置 1	1
SUBFWB	f,d,a	带借位地将 WREG 的内容减去文件寄存器的内容	1
SUBWF	f,d,a	将文件寄存器的内容减去 WREG 寄存器的内容	1
SUBWFB	f,d,a	带借位地将文件寄存器的内容减去 WREG 寄存器的内容	1
SWAPF	f,d,a	对文件寄存器的内容做半字节交换	1
TSTFSZ	f,a	测试文件寄存器，若为 0 则跳转	1
XORWF	f,d,a	将文件寄存器的内容和 WREG 寄存器的内容做异或运算	1

678

请看下面的例子。

当 d=0 且 a=0 时：

```
MyReg   SET   0x20          ;loc 20H for MyReg
MOVLW  0x45                 ;WREG = 45H
MOVWF  MyReg                ;MyReg = 45H
MOVLW  0x23                 ;WREG = 23H
ADDWF  MyReg                ;WREG = 68H (45H + 23H = 68H)
```

在上面的例子中，最后一条指令还可以写成“ADD MyReg,0,0”的形式。

当 d=1 且 a=0 时：

```
MyReg SET  0x20          ;loc 20H for MyReg
MOVLW 0x45               ;WREG = 45H
MOVWF MyReg              ;MyReg = 45H
MOVLW 0x23               ;WREG = 23H
ADDWF MyReg,F       ;MyReg = 68H (45H + 23H = 68H)
```

在上面的例子中，最后一条指令也可以写成“ADD MyReg,F,0”或者“ADDWF MyReg,1,0”。对于前面介绍的 MPLAB 仿真器，上面格式的指令是相同的。注意，指令“ADDWF MyReg,F”使用 F 代替了 1 的位置。

为了使用访问存储区以外的其他存储区，必须先对 BSR 寄存器赋值。下面的程序用到了存储区 2(RAM 地址范围为 200～2FFH)的一个地址。

当 d=0 且 a=1 时：

```
MyReg SET 0x30 ;set aside location 30H for MyReg
MOVLB 0x2      ;use Bank 2 (address loc 230H)
MOVLW 0x45               ;WREG = 45H
MOVWF MyReg,1            ;MyReg = 45H (loc 230H)
MOVLW 0x23               ;WREG = 23H
ADDWF MyReg,1  ;WREG = 68H (add loc 230H to W)
```

当 d=1 且 a=1 时：

```
MyReg  SET  0x20         ;loc 20H for MyReg
MOVLB 0x4                ;use bank 4
MOVLW 0x45               ;WREG = 45H
MOVWF MyReg              ;MyReg = 45H (loc 420H)
MOVLW 0x23               ;WREG = 23H
ADDWF MyReg,F,1          ;MyReg = 68H (loc 420)
```

679

寄存器间接寻址模式使用 FSRx 作为 RAM 地址指针。PIC18 有 3 个寄存器(FSR0、FSR1 和 FSR2)可以用作指针。

例如：

```
ADDWF POSTINC0 ;add to W data pointed to by FSR0,
               ;also increment FSR0

ADDWF POSTINC1 ;add to W data pointed to by FSR1
               ;also increment FSR1
```

更详尽的信息请参阅第 6 章例 6-6。

4. 表格处理指令

表格处理指令用来读取 PIC18 程序 ROM 中的固定数据，如表 A-4 所示。当程序 ROM 为闪存时，指令还可以用来向闪存写入数据。第 14 章已经详细讨论了 TBLRD 和 TBLWRT 两条表格处理指令，还介绍了怎样使用表格读写指令来访问 EEPROM。

表 A-4 表格处理指令(来自 Microchip 的数据表)

伪指令,操作数	描 述	周 期
	将数据传送出/入程序存储器的操作	
TBLRD*	表格读取	2
TBLRD*+	表格读取,之后加 1	2
TBLRD*-	表格读取,之后减 1	2
TBLRD+*	表格读取,之前加 1	2
TBLWT*	表格写入	2
TBLWT*+	表格写入,之后加 1	2
TBLWT*-	表格写入,之后减 1	2
TBLWT+*	表格写入,之前加 1	2

5. 控制指令

诸如分支和调用的控制指令主要用于对程序流程进行控制,如表 A-5 所示。要特别注意控制指令的目标地址。某些分支指令的目标地址,如 BZ(若为 0 则跳转),是不能超出当前地址的 128 字节范围的。CALL 指令允许 PIC18 调用 2M 范围的 ROM 空间中任何位置的子例程。下一节将详细讨论这个问题。

680

表 A-5 控制指令(来自 Microchip 的数据表)

伪指令	操作数	描 述	周 期
		控制操作指令	
BC	n	如果有进位/借位,跳转	1
BN	n	如果为负数,跳转	1
BNC	n	如果没有进位/借位,跳转	1
BNN	n	如果为正数,跳转	1
BNOV	n	如果没有溢出,跳转	1
BNZ	n	如果不为零,跳转	1
BOV	n	如果溢出,跳转	1
BRA	n	无条件跳转	2
BZ	n	如果为零,跳转	1
CALL	n,s	调用子例程 第一个字 第二个字	2
CLRWDT	—	清零监视定时器	1
DAW	—	对 WREG 的内容做十进制校正操作	1
GOTO	n	转向地址 第一个字 第二个字	2
NOP	—	空操作	1
POP	—	弹出栈顶(TOS)	1
PUSH	—	压入栈顶(TOS)	1
RCALL	n	相对调用	2
RESET		软件设备复位	1
RETFIE	s	中断返回	2
RETLW	k	返回 WREG 中的立即数	2
RETURN	s	从子例程返回	2
SLEEP	—	进入待机模式	1

681

A.2 PIC18 指令集

本节将简要介绍 PIC18 的每一条指令,并给出一些例子。

ADDLW K　　将立即数加至 WREG 寄存器

功能:将立即数 *k* 加到 WREG 寄存器中
语法: ADDLW k

该指令将立即数 *k* 和 WREG 寄存器的内容相加,然后把结果存放回 WREG 寄存器。因为 WREG 是一个 1 字节的寄存器,所以操作数 *k* 也必须是 1 字节的。

ADD 指令对于有符号数和无符号数都适用,下面分别加以讨论。关于有符号数的讨论请参阅第 5 章。

1. 无符号数加法

在进行无符号数加法时,状态位 C、DC、Z、N 和 OV 都有可能被改变。这些标志位中最重要的是标志位 C。当 8 位(D0~D7)操作中 D7 位向更高位产生进位时,C 就会被置 1。

例如:

```
MOVLW 0x45          ;WREG = 45H
ADDLW 0x4F      ;WREG = 94H (45H + 4FH = 94H)
                    ;C = 0
```

例如:

```
MOVLW   0xFE          ;WREG = FEH
ADDLW   0x75          ;WREG = FE + 75 = 73H
                      ;C = 1
```

例如:

```
MOVLW   0x25          ;WREG = 25H
ADDLW   0x42      ;WREG = 67H (25H + 42H = 67H)
                      ;C = 0
```

注意,在上面的例子中忽略了 OV 标志位。虽然 ADD 指令确实会影响 OV 标志位,但是在有符号数运算中关注 OV 标志位才有意义。下面介绍有符号数的加法。

2. 有符号数和负数

在有符号数的加法中,要特别留意溢出标志位 OV,因为它表明加法运算的结果是否出错。在有符号操作中,有两条规则来设定标志位 OV 的状态。当以下的情况出现时,溢出标志位 OV 会被置 1:

(1) 如果 D6 位向 D7 位有进位,且 D7 位没有进位;

(2) 如果 D7 位有进位,且 D6 位没有向 D7 位的进位。

注意,当 D6 位和 D7 位都产生进位时,OV=0。

例如:

```
MOVLW +D'8'     ;W = 0000 1000
ADDLW +D'4'     ;W = 0000 1100 OV = 0,
                ;C = 0, N = 0
```

注意,N=D7=0 是因为结果是正数;OV=0 是因为 D6 位和 D7 位都没有进位。又因为

OV=0,所以结果是正确的[(+8)+(+4)=(+12)]。

例如:

```
MOVLW +D'66'     ;W = 0100 0010
ADDLW +D'69'     ;W = 1000 0101 = -121
ADDWF            ;W = 1000 0111 = -121
      ;(INCORRECT) C = 0, N = D7 = 1, OV = 1
```

在上面的例子中,正确的结果应该是+135[(+66)+(+69)=(+135)],但是得出的结果却是−121。OV=1 就是运算结果错误的标志。注意,N=1 是因为结果是负数;OV=1 是因为 D6 位向 D7 位有进位,且 C=0。

例如:

```
MOVLW -D'12'    ;W = 1111 0100
ADDLW +D'18'    ;W = W + (+0001 0010)
                ;W = 0000 0110 (+6) correct
                ;N = 0, OV = 0, and C = 1
```

注意,上面的结果是正确的(OV=0),因为 D6 位和 D7 位都产生了进位。

例如:

```
MOVLW -D'30'    ;W = 1110 0010
ADDLW +D'14'    ;W = W + 0000 1110
                ;W = 1111 0000 (-16, CORRECT)
                ;N = D7 = 1, OV = 0, C = 0
```

这里,OV=0 是因为 D7 位和 D6 位都没有产生进位。

例如:

```
MOVLW -D'126'   ;W = 1000 0010
ADDLW -D'127'   ;W = W + 1000 0001
           ;W = 0000 0011 (+3, INCORRECT)
                ;D7 = N = 0, OV = 1
```

这里 C=1 是因为 D7 位产生了进位,且 D6 位没有向 D7 位进位。

从上面的讨论中可以得出结论:进位标志位 C 在任何加法中都是很重要的,溢出标志位 OV 对于有符号数加法尤其重要,因为它表明结果的正确与否。正如在指令 DAW 中所看到的一样,DC 标志位用于 BCD 数的加法。

683

ADDWF　　将 WREG 寄存器的内容与 f 寄存器的内容相加

功能:将 WREG 寄存器的内容和文件寄存器的内容相加

语法:ADDWF f, d, a

该指令把文件寄存器的内容加到 WREG 寄存器中,运算结果放入 WREG(若 d=0)或者文件寄存器(若 d=1)。

ADDWF 指令既可以用于有符号数的操作,也可以用于无符号数的操作。

例如:

```
MyReg  SET  0x20     ;loc 20H for MyReg
MOVLW 0x45           ;WREG = 45H
MOVWF MyReg          ;MyReg = 45H
MOVLW 0x4F           ;WREG = 4FH
ADDWF MyReg     ;WREG = 94H (45H + 4FH = 94H)
                     ;C = 0
```

若把结果放入文件寄存器,则有下面的例子:

```
MyReg   SET   0x20      ;loc 20H for MyReg
MOVLW  0x45             ;WREG = 45H
MOVWF  MyReg            ;MyReg = 45H
MOVLW  0x4F             ;WREG = 4FH
ADDWF  MyReg,F          ;MyReg = 94H
                    ;(45H + 4FH = 94H), C = 0
```

对于 a=0 和 a=1 的情况,请参阅 A.1 节。

ADDWFC　　将 WREG 寄存器的内容和进位标志位加到文件寄存器中

功能:将 WREG 寄存器的内容和进位标志位加到文件寄存器中

语法:ADDWFC f, d, a

该指令把 WREG 的内容和 C 标志位加到 fileReg 中(目的地址=WREG+C+fileReg)。如果在这条指令有 C=1,那么 C=1 也会被加到目的地址中。如果在这条指令前有 C=0,那么源操作数被加到目的地址,且再加 0。这条指令可用于多字节的加法。例如将 25F2H 和 3189H 相加,可以使用 ADDWFC 指令,如下例所示。

例如,当 d=0 时:

假设在 RAM 地址 0x10 和 0x11 中有如下的数据:

0x10=(F2)　　　0x11=(25)

```
Reg_L     SET  0x10 ;loc 0x10 for Reg_L
Reg_H     SET  0x11 ;loc 0x11 for Reg_H
BCF  STATUS,C       ;make carry = 0
MOVLW 89H           ;WREG = 89H
ADDWFC Reg_L,1 ;Reg_L = 89H + F2H + 0 = 7BH
               ;and C = 1
MOVLW 0x31     ;WREG = 31H
ADDWFC Reg_H,1 ;Reg_H = 31H + 25H + 1 = 57H
```

684

因此,运算的结果为:

```
 25F2H
+3189H
 577BH
```

ANDLW　　将立即数和 WREG 的内容做逻辑与运算

功能:将立即数 *k* 和 WREG 的内容做逻辑与运算

语法:ANDLW　k

该指令对 WREG 寄存器的内容和立即操作数按位做逻辑与操作,然后把结果放入 WREG 寄存器。

A	*B*	*A*逻辑与*B*
0	0	0
0	1	0
1	0	0
1	1	1

例如:

```
MOVLW   0x39    ;W = 39H
ANDLW   0x09    ;W = 39H ANDed with 09

39H  0011 1001
09H  0000 1001
09H  0000 1001
```

例如：

```
MOVLW 32H   ;W = 32H        32H  0011 0010
ANDLW 50H   ;AND W with     50H  0101 0000
            ;(W = 10H)      10H  0001 0000
```

ANDWF　　　　将 WREG 的内容同 fileReg 的内容做逻辑与运算

功能：字节变量的逻辑与操作

语法：ANDWF f，d，a

该指令对文件寄存器的值和 WREG 寄存器的值进行按位地逻辑与操作，然后把结果放入 WREG 寄存器(若 d=0)或者 fileReg 寄存器(若 d=1)。

例如：

```
MyReg  SET 0x40;set MyReg loc at 0x40
MOVLW  0x39     ;W = 39H
MOVWF  MyReg    ;MyReg = 39H
MOVLW  0x09
ANDWF  MyReg    ;39H ANDed with 09 (W = 09)

39H  0011 1001
09H  0000 1001
09H  0000 1001
```

再例如：

```
MyReg  SET 0x40;set MyReg loc at 0x40
MOVLW 0x32      ;W = 32H
MOVWF MyReg     ;MyReg = 32H
MOVLW 0x0F      ;WREG = 0FH
ANDLW MyReg     ;32H ANDed with 0FH (W = 02)

32H  0011 0010
0FH  0000 1111
02H  0000 0010
```

若要把把结果放入 fileReg，如下面的例子所示：

```
MyReg  SET 0x40;set MyReg loc at 0x40
MOVLW 0x32      ;W = 32H
MOVWF MyReg     ;MyReg = 32H
MOVLW 0x50      ;WREG = 50H
ANDLW MyReg,F   ;MyReg = 10H WREG = 50H
```

下面的指令用来清除(屏蔽)输出端口的指定位，假设所有端口都是输出方式：

```
MOVLW 0xFE
ANDWF PORTB,F   ;mask PORTB.0 (D0 of Port B)
MOVLW 0x7F
ANDWF PORTC,F   ;mask PORTC.7 (D7 of Port C)
MOVLW 0xF7
ANDWF PORTD,F   ;mask PORTD.3 (D3 of Port D)
```

分支条件

功能：条件分支(跳转)

在这类分支(跳转)指令里，在满足一定条件时，控制会转到目标地址。下面是有关标志位

685

的分支指令列表：

BC	若有进位/借位则分支	当 C=1 时跳转
BNC	若无进位/借位则分支	当 C=0 时跳转
BZ	若为 0 则分支	当 Z=1 时跳转
BNZ	若非 0 则分支	当 Z=0 时跳转
BN	若为负则分支	当 N=1 时跳转
BNN	若非负则分支	当 N=0 时跳转
BOV	若溢出则分支	当 OV=1 时跳转
BNOV	若无溢出则分支	当 OV=0 时跳转

注意，所有的"分支条件"指令都是短跳转的，即目标地址向后不能超过－128 字节，向前不能超过 127 字节。换而言之，目标地址不能超出当前 PC 的－128～127 字节的范围。那么当程序需要使用"分支条件"指令跳转到超出－128～127 范围的目标地址时，该怎么办呢？解决问题的方法是把"分支条件"和无条件跳转指令 GOTO 结合起来使用，如下所示。

686

```
          ORG   0x100
          MOVLW 0x87      ;WREG = 87H
          ADDLW 0x95      ;C = 1 after addition
          BNC   NEXT      ;branch if C = 0
          GOTO  OVER      ;target more than 128 bytes away
NEXT:     ...
          ...
          ...
          ORG 0x5000
OVER:     MOVWF PORTD
```

BC　　若 C=1 则跳转

功能：当进位/借位标志位为 1 时跳转

语法：BC 目标地址

该指令在 C=1 时跳转。

例如：

```
     MOVLW 0x0        ;WREG = 0
BACK ADDLW 0x1        ;add 1 to WREG
     BC    EXIT       ;exit if C = 1
     BRA   BACK       ;keep doing it
EXIT .....
     .....
```

注意，这是一条双字节的指令。因此，目标地址不能超过当前程序计数器的－128～127 字节范围。关于这个问题的深入讨论请参阅分支条件部分。

BCF　　对 fileReg 的位清零

功能：将 fileReg 的指定位清零

语法：BCF f,b,a

该指令可以将给定文件寄存器的某一位清零。这些位可以是端口、寄存器或者 RAM 地址的可直接寻址位。下面是一些关于指令格式的例子：

```
BCF   STATUS,C   ;C = 0
BCF   PORTB,5    ;CLEAR PORTB.5 (PORTB.5 = 0)
BCF   PORTC,7    ;CLEAR PORTC.7 (PORTC.7 = 0)
BCF   MyReg,1    ;CLEAR D1 OF File Register MyFile
```

687

BN　　若 N=1 则跳转

功能：若负数标志位=1 则跳转

语法：BN 目标地址

该指令实现在 N=1 时的跳转。它可用在有符号数的加法中，如 ADDLW 指令。注意，这是一条 2 字节的指令；因此，目标地址不能超过程序计数器的−128～127 字节范围。关于这个问题的深入讨论请参阅分支条件部分。

BNC　　若无进位则跳转

功能：若进位标志位等于零则跳转

语法：BNC 目标地址

该指令将检查 C 标志位，若为 0 则会跳转(分支)到目标地址上。

例如：计算 F6H、98H 和 8AH 的和。把进位保存在寄存器 C_Reg 中。

```
    C_Reg SET 0x20 ;set aside loc 0x20 for carries

            MOVLW  0x0      ;W = 0
            MOVWF  C_Reg    ;C_Reg = 0
            ADDLW  0xF6
            BNC    OVER1
            INCF   C_Reg,F
OVER1:      ADDLW  0x98
            BNC    OVER2
            INCF   C_Reg,F
OVER2:      ADDWF  0x8A
            BNC    OVER3
            INCF   C_Reg
OVER3:
```

注意，这是一条 2 字节的指令；因此，目标地址不能超过程序计数器的−128～127 字节范围。关于这个问题的深入讨论请参阅分支条件部分。

BNN　　若不为负数则跳转

功能：若负数标志位=0 则跳转

语法：BNN 目标地址

该指令实现在 N=0 时的跳转。它可用在有符号数的加法中，如 ADDLW 指令。注意，这是一条 2 字节的指令；因此，目标地址不能超过当前程序计数器的−128～127 字节范围。关于这个问题的深入讨论请参阅分支条件部分。

688

BNOV　　若没有溢出则跳转

功能：若溢出标志位=0 则跳转

语法:BNOV 目标地址

该指令实现在 OV=0 时的跳转。它可用在有符号数的加法中,如 ADDLW 指令。注意,这是一条 2 字节的指令;因此,目标地址不能超过当前程序计数器的－128 到 127 字节范围。关于这个问题的深入讨论请参阅分支条件部分。

BNZ　　若非零则跳转

功能:若零标志位=0 则跳转

语法:BNZ 目标地址

该指令在 Z=0 时跳转。

例如:

```
      CLRF TRISB       ;PORTB as output
      CLRF PORTB       ;clear PORTB
OVER  INCF PORTB,F     ;INC PORTB
      BNZ  OVER        ;do it until it becomes zero
```

例如:将 7 加到 WREG 中,执行 5 次。

```
      COUNTER   SET  0x20 ;loc 20H for COUNTER
      MOVLW  0x5        ;WREG = 5
      MOVWF  COUNTER    ;COUNTER = 05
      MOVLW  0x0        ;WREG = 0
OVER  ADDLW  0x7        ;add 7 to WREG
      DECF   COUNTER,F  ;decrement counter
      BNZ    OVER       ;do it until counter is zero
```

注意,这是一条 2 字节的指令;因此,目标地址不能超过当前程序计数器的－128～127 字节范围。关于这个问题的深入讨论请参阅分支条件部分。

BOV　　若溢出则跳转

功能:若溢出标志位=1 则跳转

语法:BOV 目标地址

该指令实现在 OV=1 时的跳转。它可用在有符号数的加法中,如 ADDLW 指令。注意,这是一条 2 字节的指令;因此,目标地址不能超过当前程序计数器的－128～127 字节范围。关于这个问题的深入讨论请参阅分支条件部分。

689

BRA　　无条件跳转

功能:无条件跳转

语法:BRA 目标地址

BRA 表示“分支(branch)”。它让程序无条件地转移到目标地址继续执行。这条指令的目标地址必须在程序存储器的 1 KB 范围内。这是一条 2 字节的指令。前 5 位表示操作码,其余位是有符号偏移量,用于同 BRA 下面指令的 PC(程序计数器)相加得到新的目标地址。因此,在该指令的跳转中,目标地址必须在 BRA 下一条指令的程序计数器的－1024～＋1023 字节范围内,因为 11 位的地址范围是－1024～＋1023。这个地址通常称为相对地址,因为目标地址是相对于程序计数器(PC)的－1024～＋1023 字节范围。

BSF 对 fileReg 做置位操作

功能：置位
语法：BSF f, b, a

该指令对文件寄存器的指定位置高电平。该位可以是端口、寄存器或者 RAM 地址的任何可直接寻址位。

例如：

```
BSF   PORTB,3    ;make PORTB.3 = 1
BSF   PORTC,6    ;make PORTC.6 = 1
BSF   MyReg,2    ;make bit D2 of MyReg = 1
BSF   STATUS,C   ;set Carry Flag C = 1
```

BTFSC fileReg 位测试，若为零则跳过

功能：如果所测试的指定位为 0，跳过下一条指令
语法：BTFSC f, b, a

该条指令用来测试指定的位，并且在该位为低电平时跳过下一条指令。该位可以是 PIC18 的 RAM、端口或者寄存器的可直接寻址位。

例如：监视 PORTB.5，当它变为低电平时，把 55H 放入 WREG 中。

```
          BSF TRISB,5      ;make PORTB.5 an input bit
HERE      BTFSC PORTB,5    ;skip if PORTB.5 = 0
          BRA   HERE
          MOVLW 0x55       ;because PORTB.5 = 0,
                           ;put 55H in WREG
```

690

例如：查看 WREG 的值是否为偶数。如果是，把它变成奇数。

```
          BTFSC WREG,0     ;skip if it is odd
          BRA   NEXT
          ADDLW 0x1        ;it is even, make it odd
NEXT:     ...
```

BTFSS fileReg 位测试，若为 1 则跳过

功能：如果所测试的指定位为 1，跳过下一条指令
语法：BTFSS f, b, a

该指令用来测试指定的位，并且在该位为高电平时跳过下一条指令。该位可以是 PIC18 的 RAM、端口或者寄存器的可直接寻址位。

例如：监视 PORTB.5，当它变为高电平时，把 55H 放入 WREG 中。

```
          BSF TRISB,5      ;make PORTB.5 an input bit
HERE      BTFSS PORTB,5    ;skip if PORTB.5 = 1
          BRA   HERE
          MOVLW 55H    ;because PORTB.5 = 0 WREG = 55H
```

例如：查看 WREG 的值是否为奇数。如果是，把它变成偶数。

```
          BTFSS WREG,0        ;skip if it is even
          BRA   NEXT
          ADDLW 0x01          ;it is even, make it odd
NEXT:     ...
```

BTG 将 fileReg 的位翻转

功能：位翻转(取反)

语法：BTG f, b, a

该指令将对二进制位取反。该位可以是 PIC18 的任何可位寻址的地址。

例如：

```
            BCF TRISB,0     ;make PORTB.0 an output
AGAIN       BTG PORTB,0     ;complement PORTB.0 bit
            BRA AGAIN       ;continuously forever
```

例如：触发 PORTB.7 150 次。

```
COUNTER     SET   0x20  ;loc 20H for COUNTER
      MOVLW  'D'150     ;WREG = 150
      MOVWF  COUNTER    ;COUNTER = 150
      BCF    TRISB,7    ;make PORTB.7 an output
OVER  BTG    PORTB.7    ;toggle PORTB.7
      DECF   COUNTER,F  ;decrement and put it in
                        ;COUNTER
      BNZ    OVER       ;do it 150 times
```

691

BZ 若为零则跳转

功能：若 Z=1 则跳转

语法：BZ 目标地址

例如：一直检查 PORTB 是否为 99H。

```
        SETF TRISB      ;port B as input
   BACK MOVFW PORTB     ;get PORTB into WREG
        SUBLW 0x99      ;subtract 99H from it
        BZ   EXIT       ;if 0x99, exit
        BRA  BACK       ;keep checking
        ...
EXIT:   ...
```

例如：翻转 PORTB 150 次。

```
MyReg      SET   0x40 ;loc 40H for MyReg
     SETF  TRISB      ;port B as output
     MOVLW D'150'     ;WREG = 150
     MOVWF MyReg
BACK COMF  PORTB      ;toggle PORTB
     DECF  MyReg,F    ;decrement MyReg
     BZ    EXIT       ;if MyReg = 0, exit
     BRA   BACK       ;keep toggling
     ...
EXIT: ...
```

注意，这是一条 2 字节的指令；因此，目标地址不能超过程序计数器的 −128～127 字节范围。关于这个问题的深入讨论请参阅分支条件部分。

CALL 调用指令

功能：子例程的转移控制

语法:CALL　k, s　　;这里 s 用于快速程序切换。

CALL 指令是一条 4 字节的指令。其中,前 12 位用作操作码,其余的 20 位用作地址。一个 20 位的地址允许访问 PIC18 的 2MB ROM 空间里的任何目标地址。如果是调用一个子例程,PC 寄存器(里面含有 CALL 指令的下一条指令地址)将被压栈,而栈指针(SP)会自动加 1。然后,程序计数器会读取新的地址,控制转到子例程。在这个过程结束时,RETURN 指令执行,PC 出栈,控制返回到 CALL 的下一条指令。

注意,CALL 是一条 4 字节指令,12 位操作码,其余 20 位表示一个 20 位的偶地址,因为所有的 PIC18 指令都是 2 字节的,地址最低位 A0 的默认值为 0,用来保证 CALL 指令不会落在目标地址的中间。CALL 指令的 20 位地址提供了 A20~A1 的地址和 A0=0,使用这 21 位的地址就可以访问 PIC18 的 2MB 地址空间的任何地方。

692

对于"CALL k, s"指令,这里有两个选择:s=0 和 s=1。当 s=0 时,它只是简单调用一个子例程。当 s=1 时,在调用子例程的同时,还要求 CPU 保存内部缓冲器(影子寄存器)中的 3 个主要寄存器 WREG、STATUS 和 BSR,以实现程序切换。这种快速程序切换只能用在主要的子例程中,因为影子寄存器的深度仅仅为 1。也就是说,因为 s=1,所有快速程序切换是没有嵌套的。请看下面的例子:

```
          ORG   0x0
MAIN      .....
          .....
          .....
          CALL  M_SUB,1      ;call and save the registers
          MOVLW 0x55 ;address of this instruction is saved on stack
          ....
;-------------------------
          ORG   0x2000
M_SUB     ......
          .....
          CALL  Y_SUB        ;we cannot use CALL Y_SUB,1
          MOVLW 0xAA;address of this instruction is saved on stack
          .....
          .....
          RETURN,1      ;return to caller and restore the registers
                        ;notice the s = 1 for RETURN
;-----------------------------------
          ORG 0x3000
Y_SUB
          .....
          .....
          RETURN
;-------------------------------
          END
```

如上面的例子所示,RETURN 指令也有两个选择:s=0 和 s=1。如果在 CALL 里使用

了 s=1,那么在 RETURN 里也要用 s=1。注意,指令"CALL 目标地址"里没有出现数字,编

693

译器默认它为 s=0。相似地,RETURN 里也没有出现数字,编译器也把它默认为 s=0。

CLRF 清零 fileReg

功能:清零

语法:CLRF f, a

该指令将对 fileReg 的整个字节清零。寄存器的所有位都会被置 0。

例如:

```
MyReg     SET  0x20 ;loc 20H for MyReg
     CLRF MyReg      ;clear MyReg
     CLRF TRISB ;clear TRISB (make PORTB output)
     CLRF PORTB      ;clear PORTB
     CLRF TMR01L     ;TMR0L = 0
```

注意,这条指令的结果只能放在 fileReg 中,而不能选择 WREG 作为目的地址。

CLRWDT

功能:监视定时器清零

语法:CLRWDT

该指令对监视定时器清零。

COMF 对 fileReg 取反

功能:对 fileReg 的内容取反

语法:COMF f, d, a

该指令对给定的 fileReg 执行取反操作。所得的结果为寄存器内容的反码;即 0 变成 1,1 变成 0。结果可以保存在 WREG(若 d=0)或者 fileReg(若 d=1)中。

例如:

```
           MOVLW 0x0       ;WREG = 0
           MOVWF TRISB     ;Make PORTB an output port.
           MOVLW 0x55      ;WREG = 01010101
           MOVWF PORTB
AGAIN      COMF PORTB,F    ;complement (toggle) PORTB
           CALL DELAY
           BRA  AGAIN ;continuously (notice WREG = 55H)
```

例如:

694

```
MyReg   SET 0x40;set MyReg loc at 0x40
MOVLW   0x39     ;W = 39H
MOVWF   MyReg    ;MyReg = 39H
COMPF   MyReg,F ;MyReg = C6H and WREG = 39H ...
```

这里,39H(0011 1001)变成了 C6H(1100 0110)。

再例如:

```
MyReg  SET 0x40;set MyReg loc at 0x40
MOVLW 0x55       ;W = 55H
MOVWF MyReg      ;MyReg = 55H
COMPF MyReg,F    ;MyReg AAH, WREG = 55H
```

这里，55H(0101 0101)变成 AAH(1010 1010)。

例如：翻转 PORTB 150 次。

```
COUNTER    SET  0x40  ;loc 40H for COUNTER
     SETF TRISB       ;port B as output
     MOVLW D'150'     ;WREG = 150
     MOVWF COUNTER    ;COUNTER = 150
     MOVLW 0x55       ;WREG = 55H
     MOVWF PORTB
BACK COMF PORTB,F     ;toggle PORTB
     DECF COUNTER,F   ;decrement COUNTER
     BNZ  BACK ;toggle until counter becomes 0
```

若要把结果放在 WREG 中，程序代码如下面的例子所示：

```
MyReg   SET  0x40     ;set MyReg loc at 0x40
     MOVLW   0x39     ;W = 39H
     MOVWF   MyReg    ;MyReg = 39H
     COMPF   MyReg    ;MyReg = 39H and WREG = C6H
```

再例如：

```
MyReg   SET     0x40  ;set MyReg loc at 0x40
     MOVLW 0x55       ;W = 55H
     MOVWF MyReg      ;MyReg = 55H
     COMPF MyReg      ;WREG = AA and MyReg 55H SETF
```

CPFSEQ　　比较 fileReg 和 WREG 的内容，若相等(F=W)则跳过

功能：比较 fileReg 的内容和 WREG 的内容，若相等则跳过下一条指令

语法：CPFSEQ f, a

该指令对 fileReg 的字节内容和 WREG 寄存器的字节内容进行比较。如果它们相等，就跳过下一条指令。

例如：一直监视 PORTB 的值是否为 99H。仅当 PORTB 的值为 99H 时跳出监视。

```
     SETF   TRISB     ;PORTB an input port
     MOVLW 0x99       ;WREG = 99h
BACK CPFSEQ PORTB     ;skip if PORTB has 0x99
     BRA BACK         ;keep monitoring
```

695

注意，只有当 fileReg 的值和 WREG 的值相同时，CPFSEQ 才会跳过下一条指令。

CPFSGT　　比较 fileReg 和 WREG 的内容，若 F>W 则跳过

功能：比较 fileReg 的内容和 WREG 的内容，如果 fileReg>WREG，跳过下一条指令

语法：CPFSGT f, a

该指令对 fileReg 的字节值和 WREG 寄存器的字节值进行比较。如果 fileReg 的值大于 WREG 的值，就跳过下一条指令。

例如：一直监视 PORTB 的值是否为 99H。仅当 PORTB 的值大于 99H 时跳出监视。

```
          SETF  TRISB     ;PORTB an input port
          MOVLW 0x99      ;WREG = 99H
BACK      CPFSGT PORTB    ;skip if PORTB > 99H
          BRA BACK        ;keep monitoring
```

注意,只有当fileReg的值大于WREG的值时,CPFSGT才会跳过下一条指令。

CPFSLT　　比较fileReg和WREG的内容,若F<W则跳过

功能:比较fileReg的内容和WREG的内容,若fileReg<WREG则跳过下一条指令

语法:CPFSLT f, a

该指令对fileReg的字节值和WREG寄存器的字节值进行比较。如果fileReg的值小于WREG的值,就跳过下一条指令。

例如:一直监视PORTB的值是否为99H。仅当PORTB的值小于99H时跳出监视。

```
          SETF   TRISB      ;PORTB an input port
          MOVLW  0x99       ;WREG = 99H
BACK:     CPFSEQ PORTB      ;skip if PORTB < 99H
          BRA BACK          ;keep monitoring
```

注意,只有当fileReg的值小于WREG的值时,CPFSLT才会跳过下一条指令。

DAW　　十进制校正

功能:在执行加法运算后,对WREG的值进行十进制校正

语法:DAW

该指令用在BCD加法之后,把运算的结果转换为BCD码。在下面两种情况下,数据将被DAW校正。

(1) 当WREG低4位的值大于9或者DC=1时,低4位会加6。

696

(2) 当WREG高4位的值大于9或者C=1时,高4位也会加6。

例如:

```
MOVLW 0x47    ;WREG = 0100 0111
ADDLW 0x38    ;WREG = 47H + 38H = 7FH,
              ;invalid BCD
DAW      ;WREG = 1000 0101 = 85H, valid BCD

  47H
+ 38H
  7FH  (invalid BCD)
+  6H  (after DAW)
  85H  (valid BCD)
```

在上面的例子中,因为低半字节大于9,DAW就加6到WREG中。如果低半字节小于9但DC=1,低半字节也会加6。请看下面的例子:

```
MOVLW 0x29     ;WREG = 0010 1001
ADDLW 0x18     ;WREG = 0100 0001 INCORRECT
DAW       ;WREG = 0100 0111 = 47H VALID BCD

  29H
+ 18H
  41H  (incorrect result in BCD)
+  6H
  47H  correct result in BCD
```

对高半字节的十进制校正,也有相同的情况。请看下面的例子:

```
MOVLW 0x52      ;WREG = 0101 0010
ADDLW 0x91      ;WREG = 1110 0011 INVALID BCD
DAW             ;WREG = 0100 0011 AND C = 1

 52H
+ 91H
 E3H       (invalid BCD)
+ 6        (after DAW, adding to upper nibble)
143H       valid BCD
```

相似地，如果高半字节小于 9 而 C=1，就必须要进行校正。请看下面的例子：

```
MOVLW 0x94            ;W = 1001 0100
ADDLW 0x91            ;W = 0010 0101 INCORRECT
DAW               ;W = 1000 0101, VALID BCD
                      ;FOR 85, C = 1

 94H
+ 91H
1 2 5H      (incorrect BCD)
+ 6         (after DAW, adding to upper nibble)
1 8 5H
```

另外，还有可能出现高半字节和低半字节都需要加 6 的情况。请看下面的例子：

```
MOVLW   0x54     ;WREG = 0101 0100
ADDLW   0x87     ;WREG = 1101 1011 INVALID BCD
DAW          ;WREG = 0100 0001, C = 1 (BCD 141)

 54H
+ 87H
 DBH  (invalid result in BCD)
+ 6 6H
1 4 1H       valid BCD
```

697

DECF　　fileReg 自减 1

功能：将 fileReg 的内容减 1

语法：DECF f, d, a

该指令将 fileReg 里的字节操作数减 1。结果可以放在 WREG(若 d=0)或者 fileReg(若 d=1)中。

例如：

```
MyReg SET 0x40 ;set aside loc 40H for MyReg
MOVLW 0x99      ;WREG = 99H
MOVWF MyReg     ;MyReg = 99H
DECF MyReg,F    ;MyReg = 98H, WREG 99H
DECF MyReg,F    ;MyReg = 97H, WREG 99H
DECF MyReg,F    ;MyReg = 96H, WREG 99H
```

再例如：翻转 PORTB 250 次。

```
COUNTER   SET  0x40 ;loc 40H for COUNTER
     SETF TRISB     ;PORTB as output
     MOVLW D'250'   ;WREG = 250
     MOVWF COUNTER  ;COUNTER = 250
```

```
      MOVLW  0x55       ;WREG = 55H
      MOVWF  PORTB
BACK  COMF  PORTB,F     ;toggle PORTB
      DECF  COUNTER,F ;decrement COUNTER
      BNZ   BACK ;toggle until counter becomes 0
```

698

若要把结果放在 WREG 中，则有如下的例子：

```
MyReg SET 0x40 ;set aside loc for MyReg
MOVLW  0x99      ;WREG = 99H
MOVWF  MyReg     ;MyReg = 99H
DECF  MyReg      ;WREG = 98H, MyReg = 99H
DECF  MyReg      ;WREG = 97H, MyReg = 99H
DECF  MyReg      ;WREG = 96H, MyReg = 99H
```

再例如：

```
MyReg   SET     0x50 ;set MyReg loc at 0x50
MOVLW   0x39    ;W = 39H
MOVWF   MyReg   ;MyReg = 39H
DECF    MyReg   ;WREG = 38H and MyReg = 39H
DECF    MyReg   ;WREG = 37H and MyReg = 39H
DECF    MyReg   ;WREG = 36H and MyReg = 39H
DECF    MyReg   ;WREG = 35H and MyReg = 39H
```

DECFSZ　　将 fileReg 减 1，若为零则跳过

功能：将 fileReg 的内容减 1，并且当 fileReg 为 0 时跳过下一条指令

语法：DECFSZ f, d, a

该指令将 fileReg 里的字节操作数减 1。若结果为 0，则跳过下一条指令的执行。

例如：翻转 PORTB 250 次。

```
COUNT      SET   0x40 ;loc 40H for COUNT
      CLRF TRISB      ;PORTB an output
      MOVLW D'250'    ;WREG = 250
      MOVWF COUNT     ;COUNT = 250
      MOVLW 0x55      ;WREG = 55H
      MOVWF PORTB

BACK  COMF PORTB,F    ;toggle PORTB
      DECFSZ COUNT,F  ;decrement COUNT and
                      ;skip if zero
      BRA  BACK ;toggle until counter becomes 0
      ...
```

DECFSNZ　　将 fileReg 减 1，若不为零则跳过

功能：将 fileReg 的内容减 1，并且当 fileReg 非 0 时跳过下一条指令

语法：DECFSNZ f, d, a

699

该指令将 fileReg 里的字节操作数减 1。若结果非 0，跳过下一条指令的执行。

例如：连续地反转 PORTB 250 次。

```
COUNT      SET   0x40 ;loc 40H for COUNT
      CLRF TRISB      ;PORTB an output
```

```
OVER MOVLW D'250'    ;WREG = 250
     MOVWF COUNT     ;COUNT = 250
     MOVLW 0x55      ;WREG = 55H
     MOVWF PORTB
BACK COMF PORTB,F    ;toggle PORTB
     DECFSNZ COUNT,F ;decrement COUNT and
                     ;skip if zero
     BRA   OVER      ;start over
     BRA   BACK ;toggle until counter becomes 0
```

GOTO　　　　　无条件转移

功能：控制无条件地跳转到新的地址

语法：GOTO k

在 PIC18 里有两条无条件分支(跳转)指令：GOTO(长跳转)和 BRA(短跳转)。下面将分别加以介绍。

(1) GOTO(长跳转)：这是一条 4B 的指令。前 12 位是操作码，后面 20 位是一个偶数目标地址。因为所有的 PIC18 指令都是 2B 的，最低地址位 A0 的默认值为 0，这样可保证 GOTO 指令不会落在目标指令的中间。GOTO 指令的 20 位地址提供了 A20～A1 的地址和 A0=0，使用这 21 位的地址就可以访问 PIC18 的 2MB 地址空间的任何地方。

(2) BRA(短跳转)：这是一条 2B 的指令。前 5 位是操作码，剩下的 11 位是有符号偏移量，用于同 BRA 下面指令的 PC(程序计数器)相加，得到目标地址。因此，在这个跳转指令中，目标地址必须是在 BRA 下一指令的程序计数器的－1024～＋1023B 范围之内，因为 11 位的地址范围是－1024～＋1023B。该地址通常称为相对地址，因为目标地址是相对于程序计数器(PC)的－1024～＋1023B 范围的。

GOTO 指令可以跳转到 PIC18 的 2MB 代码空间中的任何位置，而 BRA 只能在 1KB ROM 空间内跳转。BRA 的优点在于它只占用 2B 的程序 ROM 空间，而 GOTO 则会占用 4B。BRA 指令广泛地用于程序 ROM 容量小和引脚数目有限的芯片。

注意，GOTO 指令和 CALL 指令的区别是，CALL 指令具有返回功能，并且继续执行 CALL 下面的一条指令；而 GOTO 指令不具备返回功能。

INCF　　　　fileReg 自加 1

功能：自增量加 1

语法：INCF f, d, a

该指令将 fileReg 寄存器的字节操作数加 1。所得的结果可以放在 WREG(若 d=0)或者 fileReg(若 d=1)中。

例如：

```
MyReg SET 0x40 ;set aside loc 40H for MyReg
MOVLW 0x99     ;WREG = 99H
MOVWF MyReg
INCF MyReg,F   ;MyReg = 9AH, WREG 99H
INCF MyReg,F   ;MyReg = 9BH, WREG 99H
DECF MyReg,F   ;MyReg = 9CH, WREG 99H
```

又例如：翻转 PORTB 5 次。

```
COUNTER   SET  0x40 ;loc 40H for COUNTER
      SETF TRISB       ;PORTB as output
      MOVLW D'251'     ;WREG = 251
      MOVWF COUNTER    ;COUNTER = 251
      MOVLW 0x55       ;WREG  = 55H
      MOVWF PORTB
 BACK COMF PORTB,F     ;toggle PORTB
      INCF COUNTER,F ;INC COUNTER
      BNC  BACK ;toggle until counter becomes 0
```

若要把结果存放在 fileReg 中，则如下面的例子所示：

```
MyReg SET 0x40 ;set aside loc for MyReg
MOVLW 0x99     ;WREG = 99H
MOVWF MyReg    ;MyReg = 99H
INCF MyReg     ;WREG = 9AH, MyReg = 99H
INCF MyReg     ;WREG = 9BH, MyReg = 99H
```

再例如：

```
MyReg  SET    0x40 ;set MyReg loc at 0x40
MOVLW  0x5    ;W = 05H
MOVWF  MyReg  ;MyReg = 05H
INCF   MyReg  ;WREG = 06H and MyReg = 05H
```

INCFSZ　　将 fileReg 加 1，若为零则跳过

701

功能：自增量加 1

语法：INCFSZ f, d, a

该指令将让 fileReg 的字节操作数加 1。若结果为 0，则跳过下一条指令的执行。

例如：翻转 PORTB 156 次。

```
COUNTER  SET  0x40 ;loc 40H for COUNTER
      SETF TRISB      ;PORTB as output
      MOVLW D'156'    ;WREG = 156
      MOVWF COUNTER   ;COUNTER = 156
      MOVLW 0x55      ;WREG = 55H
      MOVWF PORTB
BACK  COMF PORTB,F    ;toggle PORTB
      INCFSZ COUNTER,F ;INC COUNTER and skip if 0
      BRA  BACK ;toggle until cou ter becomes 0
      .....
```

INCFSNZ　　将 fileReg 加 1，若不为零则跳过

功能：自增量加 1

语法：INCFSNZ f, d, a

该指令将寄存器的内容或者操作数制定的存储器中的内容加 1。若结果非 0，跳过下一条指令的执行。

例如：连续地将 PORTB 翻转 156 次。

```
COUNTER    SET  0x40 ;loc 40H for COUNTER
      SETF TRISB     ;PORTB as output
OVER  MOVLW D'156'   ;WREG = 156
      MOVWF COUNTER  ;COUNTER = 156
      MOVLW 0x55     ;WREG = 55H
      MOVWF PORTB
BACK  COMF PORTB,F   ;toggle PORTB
      INCFSNZ COUNTER,F;INC COUNTER, skip if not 0
      BRA  OVER ;start over
      BRA  BACK ;toggle until counter becomes 0
```

IORLW　　　　用立即数 k 同 WREG 的内容做逻辑或运算

功能：将 WREG 寄存器的内容和立即数 k 做逻辑或运算

语法：IORLW k

该指令对 WREG 寄存器的内容和立即数 k 做逻辑与运算，按位地进行，所得的结果放入 WREG 中。

例如：

```
MOVLW 0x30    ;W = 30H
IORLW 0x09    ;now W = 39H

39H    0011 0000
09H    0000 1001
39     0011 1001
```

A	B	A 或 B
0	0	0
0	1	1
1	0	1
1	1	1

702

又例如：

```
MOVLW 0x32      ;W = 32H
IORLW 0x50      ;(W = 72H)
32H    0011 0010
50H    0101 0000
72H    0111 0010
```

IORWF　　　　将 fileReg 同 WREG 做逻辑或运算

功能：将 fileReg 的内容和 WREG 的内容做逻辑或运算

语法：IORWF f，d，a

该指令对 WREG 寄存器的内容和 fileReg 的内容执行逻辑与运算，按位进行，所得的结果可以放在 WREG(当 d=0 时)或者 fileReg(当 d=1 时)中。

例如：

```
MyReg   SET 0x40;set MyReg loc at 0x40
MOVLW   0x39     ;WREG = 39H
MOVWF   MyReg    ;MyReg = 39H
MOVLW   0x07
IORWF   MyReg    ;39H ORed with 07 (W = 3F)

39    0011 1001
07    0000 0111
3F    0011 1111
```

又例如：

```
MyReg   SET 0x40;set MyReg loc at 0x40
MOVLW   0x5      ;WREG = 05H
MOVWF   MyReg    ;MyReg = 05H
MOVLW   0x30
IORWF   MyReg    ;30H ORed with 05 (W = 35H)

05H   0000 0101
30H   0011 0000
35H   0011 0101
```

若要把结果存放在 fileReg 中，程序代码如下面的例子所示：

703

```
MOVLW 0x30        ;W = 30H
IORWF PORTB,F     ;W and PORTB are ORed and result
                  ;goes to PORTB
```

再例如：

```
MyReg       SET   0x20
MOVLW 0x54        ;WREG = 54H
MOVWF MyReg
MOVLW 0x67        ;WREG = 67H
IORWF MyReg,F     ;OR WREG and MyReg
;after the operation MyReg = 77H

44H   0101 0100
67H   0110 0111
77H   0111 0111
```

因此，MyReg 的值为 77H，WREG 的值为 54H。

LFSR　　　　　　加载 FSR

功能：把 12 位的立即数 k 加载到 FSR 寄存器中

语法：LFSR f, k　　　；k 是 000～FFFH 之间的立即数

该指令将 12 位的立即数 k 传送到 FSR0、FSR1、FSR2 中的一个 FSR 寄存器中。如：

```
LFSR 0 , 0x200 ;FSR0 = 200H
LFSR 1 , 0x050 ;FSR1 = 050H
LFSR 2 , 0x160 ;FSR2 = 160H
```

这条指令常用于寄存器的间接寻址模式。请参阅第 6 章。

MOVF(或 MOVFW)　　　　将 fileReg 的内容复制到 WREG

功能：把 fileReg 的一个字节内容复制到 WREG 中

语法：MOVF f, d, a;

该指令常用于把数据从 fileReg 传送到 WREG。请看下面的例子：

```
CLRF      TRISC     ;PORTC output
SETF      TRISB     ;PORTB as input
MOVFW     PORTB     ;copy PORTB to WREG
ANDLW     0x0F      ;mask the upper 4 bits
MOVWF     PORTC     ;put it in PORTC
```

又例如：

```
CLRF     TRISD     ;PORTD as output
SETF     TRISB     ;PORTB as input
MOVFW    PORTB     ;copy PORTB to WREG
IORW     0x30      ;OR it with 30H
MOVWF    PORTD     ;put it in PORTD
```

这条指令还可用来把 fileReg 的内容传送回给自己，以改变状态寄存器的 N 和 Z 标志位。请看下面的例子。

704

例如：

```
MyReg SET 0x20 ;set aside loc 0x20 to MyReg
MOVLW 0x54      ;W = 54H
MOVWF MyReg     ;MyReg = 54H
MOVFW MyReg,F ;My Reg = 54, also N = 0 and Z = 0
```

MOVFF　　从 fileReg 复制到 fileReg

功能：把字节数从一个 fileReg 复制到另一个 fileReg 中

语法：MOVFF fs, fd

该指令把一个字节从源地址传送到目的地址。源地址和目的地址可以是文件寄存器、SFR 或者端口的任何位置。如：

```
MOVFF    PORTB,MyReg
MOVFF    PORTC,PORTD
MOVFF    RCREG,PORTC
MOVFF    Reg1,REG2
```

注意，这是一条 4 B 的指令，因为源地址和目的地址都占了指令的 12 位大小。也就是说，指令的 24 位都用于表示源地址和目的地址。12 位的地址允许在 PIC 18 的 4 KB RAM 空间里任意地将数据从一个地址复制到另一个地址。

MOVLB　　将 4 位的立即数传送到 BSR 的低 4 位

功能：把 4 位的立即数传送到 BSR 寄存器的低 4 位

语法：MOVLB　k　　;k 是 0～15 的值(十六进制的 0～F)

该指令用来选择一个寄存器存储区，而不是访问存储区。使用这条指令可以把一个 4 位立即数送入 BSR(存储区选择寄存器)，用以表示 PIC18 的 16 个存储区之一。换而言之，这个立即数的取值范围是 0000～1111(0～F，十六进制)。关于 MOVLB 的例子，请参阅第 6 章和本章 A.1 节。

MOVLW k　　将立即数传送给 WREG

功能：把 8 位的立即数 k 送入 WREG 中

语法：MOVLW　k　　　;k 的取值范围为 0～255(0～FF，十六进制)

例如：

```
MOVLW    0x55      ;WREG = 55H
MOVLW    0x0       ;clear WREG (WREG = 0)
MOVLW    0xC2      ;WREG = C2H
MOVLW    0x7F      ;WREG = 7FH
```

该指令和 MOVWF 常用来把常数传送至任意端口、SFR 或者 fileReg 地址。请参考下

705 面的指令,看它是如何使用的。

MOVWF 将 WREG 复制到 fileReg

功能:把 WREG 寄存器的内容复制到 fileReg 中

语法:MOVWF f, a

该指令把一个字节数据从 WREG 寄存器传送到 fileReg 中。它和指令 MOVLW 常用来把常数传送给任意的 fileReg 寄存器、SFR 或者端口地址。请看下面的例子。

例如:翻转 PORTB。

```
MOVLW     0x55        ;WREG = 55H
MOVWF     PORTB
MOVLW     0xAA        ;WREG = AAH
MOVWF     PORTB
BRA       OVER        ;keep toggling the PORTB
```

又例如:把 RAM 地址 20H 赋值为 50H。

```
MyReg SET 0x20 ;set aside the loc 0x20 for MyReg
MOVLW     0x50
MOVWF     MyReg       ;MyReg = 50H (loc 20H has 50H)
```

再例如:初始化定时器 0 的高低字节寄存器。

```
MOVLW     0x05        ;WREG = 05H
MOVWF     TMR0H       ;TMR0H = 0x5
MOVLW     0x30        ;WREG = 30H
MOVWF     TMR0L       ;TMR0L = 0x30
```

MULLW 将立即数和 WREG 相乘

功能:乘法运算 k×WREG

语法:MULLW k

该条指令用来实现无符号字节数 k 和 WREG 寄存器中的无符号数的乘法,并把 16 位的运算结果存放在寄存器 PRODH 和 PRODL 中,其中,PRODL 放低字节,而 PRODH 放高字节。

例如:

```
MOVLW 0x5        ;WREG = 5H
MULLW 0x07       ;PRODL = 35 = 23H, PRODH = 00
```

又例如:

```
MOVLW 0x0A       ;WREG = 10
MULLW 0x0F       ;PRODL = 10 x 15 = 150 = 96H
                 ;PRODH = 00
```

再例如:

706

```
MOVLW 0x25
MULLW 0x78   ;PRODL = 58H, PRODH = 11H
;because 25H x 78H = 1158H
```

再例如:

```
MOVLW D'100'     ;WREG = 100
MULLW D'200'     ;PRODL = 20H, PRODH = 4EH
            ;(100 x 200 = 20,000 = 4E20H)
```

MULWF　　将 WREG 与 fileReg 的内容相乘

功能:乘法运算 WREG×fileReg,并且把结果放入 PRODH:PROFDL 寄存器

语法:MULWF f, a

该指令将 WREG 寄存器的内容和文件寄存器里的无符号数相乘,并把 16 位的结果放在寄存器 PRODH 和 PRODL 中,其中,PRODL 放低字节,而 PRODH 放高字节。

例如:

```
MyReg      SET  0x20 ;MyReg has location of 0x20
     MOVLW  0x5
     MOVWF  MyReg     ;MyReg has 0x5
     MOVLW  0x7       ;WREG = 0x7
     MULWF  MyReg     ;PRODL = 35 = 23H, PRODH = 00
```

例如:

```
MOVLW  0x0A
MOVWF  MyReg     ;MyReg = 10
MOVLW  0x0F      ;WREG = 15
MULFW  MyReg   ;PRODL = 150 = 96H, PRODH = 00
```

例如:

```
MOVLW  0x25
MOVWF  MyReg      ;MyReg = 0x25
MOVLW  0x78       ;WREG 78H
MULWF  Myreg      ;PRODL = 58H, PRODH = 11H
                  ;(25H x 78H = 1158H)
```

例如:

```
MOVLW  D'100'     ;WREG = 100
MOVWF  MyReg      ;MyReg = 100
MOVLW  D'200'     ;WREG = 200
MULWF  MyReg      ;PRODL = 20H, PRODH = 4EH
           ;(100 x 200 = 20,000 = 4E20H)
```

NEGF　　将 fileReg 取补

功能:无操作

语法:NEGF　f, a

该指令用来对 fileReg 的值做二进制的取补运算,并把结果放回到 fileReg 中。

707

例如:

```
MyReg      SET  0x30
MOVLW  0x98       ;WREG = 0x98
MOVWF  MyReg      ;MyReg = 0x98
NEGF              ;2's complement fileReg

98H  10011000
     01100111        1's complement
  +         1
     01101000        Now FileReg = 68H
```

又例如:

```
MyReg     SET 0x10
MOVLW 0x75        ;WREG = 0x75
MOVWF MyReg       ;MyReg = 0x75
NEGF              ;2's complement fileReg

75H  01110101
     10001010        1's complement
   +        1
     10001011        Now FileReg = 8BH
```

注意,这条指令不能把结果存放到 WREG 寄存器中。

NOP　　空指令

功能:无操作

语法:NOP

该条指令不进行任何操作,并且接下来将执行下一条指令。有时候它会用作时延来消耗时钟周期。这条指令只是让 PC 指向 NOP 的下一条指令。在 PIC18 中,它是一条 2B 的指令。

POP　　出栈

功能:弹出栈顶

语法:POP

该条指令把栈顶(TOS)传送给 SP(栈指针)指向的位置,然后把它丢弃。同时将 SP 减 1。操作完成后,栈顶会变成上一次压栈之前的值。

PUSH　　压栈

功能:把 PC 值压入栈

语法:PUSH

该条指令把程序计数器(PC)的值送入栈,并使 SP 加 1。顾名思义,也就是将过去的栈顶往下压。

708

RCALL　　相对调用

功能:把控制转移到 1 KB 范围内的子例程

语法:RCALL 目标地址

PIC18 一共有两种类型的调用指令:RCALL 和 CALL。对于 RCALL 指令,目标地址在当前 PC 的 1KB 范围内。要访问 PIC18 的 2MB ROM 地址,则必须使用 CALL 指令。在调用子例程时,PC 寄存器(里面包含 RCALL 的下一条指令地址)会被压栈,栈指针(SP)会自动加 1。然后程序计数器将读取新的地址,控制转移到子例程。在程序结束时,RETURN 执行,PC 值出栈,控制返回到 RCALL 下面的一条指令。

注意,RCALL 是一条 2B 的指令,前 5 位表示操作码,剩下的 11 位表示目标子例程的地址。11 位地址的范围是 −1024～+1023。关于访问 PIC18 的 2MB ROM 空间中目标地址的问题,请参阅 CALL 指令。注意,RCALL 是一条 2B 的指令,而 CALL 是一条 4B 的指令。还要注意的是,RCALL 没有如 CALL 那样的现场保存权。

RESET **软件复位**

功能：使用软件方法复位
语法:RESET

该条指令用软件方法让 PIC18 复位。当执行完这条程序后,所有的寄存器标志位都会被强制复位。复位条件由硬件引脚 MCLR 生成。换而言之,RESET 指令是 MCLR 引脚的软件版本。

RETFIE **从中断返回**

功能:从中断返回
语法:RETFIE s

该条指令用在中断服务程序(中断处理器)的最后一行。执行该指令,栈顶会被弹出到程序计数器,从而程序在新的地址继续执行。当弹出栈顶到程序计数器(PC)后时,栈指针(SP)会自动减 1。

注意,RETURN 指令是用在子例程的最后一行,且和 CALL 和 RCALL 指令是对应的,而 RETFIE 必须用在中断服务程序(ISR)的最后一行。

709

RETLW **带立即数的返回**

功能:立即数 k 送入 WREG 中,栈顶送入 PC
语法:RETLW k

执行完该条程序后,数值 k 被送入 WREG,而栈顶被弹出到程序计数器(PC)。当弹出栈顶到程序计数器后,栈指针(SP)会自动减 1。这条指令常用于查表程序。请参阅 6.3 节。

RETURN **调用返回**

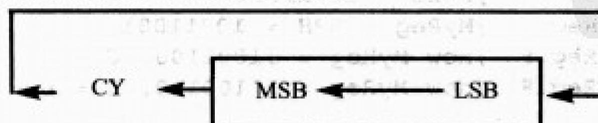
功能:从子例程返回
语法:RETURN s; s 可以是 0,可以是 1

该条指令用于从由 CALL 或者 RCALL 调用的子例程中返回。栈顶被弹出到程序计数器(PC),程序继续从新的地址执行。当栈顶被弹出到程序计数器时,栈指针(SP)自动减 1。在指令"RETURN s"中,当 s=1 时,RETURN 会再次保存现场寄存器。对于 s=1 的情况,请参阅 CALL 指令。注意,指令"RETURN 1"不能在 RCALL 调用的子例程中使用。

RLCF **带进位/借位的循环左移**

功能:带进/借位的循环左移
语法:RLCF f, d, a

该条指令将 fileReg 寄存器的二进制位循环左移 1 位。移出 fileReg 的位将被移到 C 中,而原来的 C 标志位将被移入到 fileReg 的最低位。



例如：

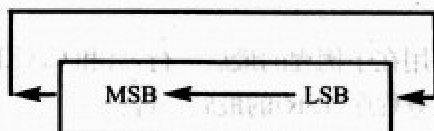
```
MyReg SET 0x30          ;set aside loc 30H for MyReg
      BCF   STATUS,C    ;C = 0
      MOVLW 0x99        ;WREG = 99H
      MOVWF MyReg       ;MyReg = 99H = 10011001
      RLCF  MyReg,F     ;now MyReg = 00110010 and
                        ;C = 1
      RLCF  MyReg,F     ;now MyReg = 01100101 and
                        ;C = 0
```

RLNCF　　　　不带进位/借位的循环左移

功能：循环左移 fileReg

语法：RLNCF f, d, a

该条指令将 fileReg 寄存器的二进制位循环左移 1 位。被移出 fileReg 的位将被移入到 fileReg 的最低位。



例如：

```
MyReg SET 0x20 ;set aside loc 20 for MyReg
MOVLW  0x69     ;WREG = 01101001
MOVWF  MyReg    ;MyReg = 69H = 01101001
RLNCF  MyReg,F  ;now MyReg = 11010010
RLNCF  MyReg,F  ;now MyReg = 10100101
RLNCF  MyReg,F  ;now MyReg = 01001011
RLNCF  MyReg,F  ;now MyReg = 10010110
```

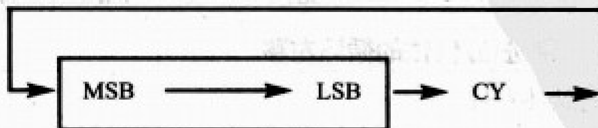
注意，经过 4 次循环移位操作后，高半字节和低半字节交换了。

RRCF　　　　带进位/借位的循环右移

功能：带进/借位的循环右移

语法：RRCF　f, d, a

该条指令将 fileReg 寄存器的二进制位右移 1 位。被移出 fileReg 的位将被移到 C 标志位中，而 C 标志位则被移入到 fileReg 的最高位。



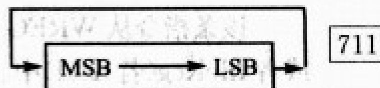
例如：

```
MyReg SET 0x20 ;set aside loc 20 for MyReg
      BSF   STATUS,C    ;C = 1
      MOVLW 0x99        ;WREG = 10011001
      MOVWF MyReg       ;MyReg = 99H = 10011001
      RRCF  MyReg,F     ;now MyReg = 11001100, C = 1
      RRCF  MyReg,F     ;now MyReg = 11100110, C = 0
```

RRNCF 不带进位/借位的循环右移

功能：将 fileReg 循环右移

语法：RRNCF f, d, a



711

该指令将 fileReg 寄存器的二进制位右移 1 位。被移出 fileReg 的位将被移入到 fileReg 的最高位。

例如：

```
MyReg SET 0x20 ;set aside loc 20H for MyReg
      MOVLW 0x66      ;WREG = 66H = 01100110
      MOVWF MyReg     ;MyReg = 66H = 01100110
      RRNCF MyReg,F   ;now MyReg = 00110011
      RRNCF MyReg,F   ;now MyReg = 10011001
      RRNCF MyReg,F   ;now MyReg = 11001100
      RRNCF MyReg,F   ;now MyReg = 01100110
```

又例如：可以使用该条指令把高半字节和低半字节交换。

```
MyReg SET 0x20 ;set aside loc 20H for MyReg
      MOVLW 0x36      ;WREG = 36H = 00110110
      MOVWF MyReg     ;MyReg = 36H = 00110110
      RRNCF MyReg,F   ;now MyReg = 00011011
      RRNCF MyReg,F   ;now MyReg = 10001101
      RRNCF MyReg,F   ;now MyReg = 11000110
      RRNCF MyReg,F   ;now MyReg = 01100011 = 63H
```

SETF 将 fileReg 全部置位

功能：置位操作

语法：SETF f, a

该条指令把 fileReg 的整个字节都置为高电平，寄存器的所有位都变成 1。

例如：

```
SETF MyReg      ;MyReg = 11111111
SETF TRISB ;TRISB = FFH,(makes PORTB input)
SETF PORTC      ;PORTC = 1111 1111
```

注意，在这条指令中，结果只能存放在 fileReg 中，WREG 不能作为目的地址。

SLEEP 进入休眠模式

功能：让 CPU 进入休眠模式

语法：SLEEP

该条指令让振荡器停止工作，并且让 CPU 进入休眠模式。它的执行将复位监视定时器(WDT)。WDT 主要和 SLEEP 指令一起使用。当 SLEEP 指令执行时，整个微控制器进入休眠模式，关掉主要的振荡器和程序计数器停止读取 SLEEP 的下一条指令。要从休眠模式唤醒，有 2 个方法：(1)通过硬件中断的外部事件，(2)内部的 WDT 中断。当通过 WDT 中断唤醒时，微控制器恢复执行 SLEEP 的下一条指令。

关于使用 WDT 的更多介绍，请登录 Microchip 公司的网站查找。

712

SUBFWB　　从 WREG 中减去 fileReg 和借位

功能:执行 WREG－fileReg－＃borrow 操作　　;＃borrow 是借位

语法:SUBFWB f, d, a

该条指令从 WREG 中减去 fileReg 的内容和借位,并且把结果存放到 WREG(若d=0)或者 fileReg(若 d=1)中。在执行减法时,CPU 的内部硬件操作如下:

(1) 对 fileReg 的内容取补码;

(2) 将所得的补码加到 WREG 中;

(3) 减去借位;

(4) 忽略借位;

(5) 检查 N(负数)标志位,看结果为正还是为负。

例如:

```
     MyReg SET 0x20 ;set aside loc 0x20 for MyReg
     BSF   STATUS,C ;make Carry = 1
     MOVLW  0x45    ;WREG 45H
     MOVWF  MyReg   ;MYReg = 45H
     MOVLW  0x23
     SUBWF  MyReg   ;WREG = 45H - 23H - 0 = 22H

 45H       0100 0101              0100 0101
-23H       0010 0011 2's comp   + 1101 1101
                Inverted carry +          0
-------                         -----------
+22H                              0010 0010
Because D7 (the N flag) is 0, the result is
positive.
```

该条指令按下面的规则来设置负数标志位:

	N	
若 WREG＞(fileReg＋＃C)	0	结果是正数
若 WREG＝(fileReg＋＃C)	0	结果是零
若 WREG＜(fileReg＋＃C)	1	结果是负数

SUBLW　　将立即数减去 WREG 的内容

功能:将立即数 k 减去 WREG 的内容(WREG=k－WREG)

语法:SUBLW　　k

该条指令用立即数 k 减去 WREG 寄存器的内容,并且把结果放入到 WREG 中。在执行减法时,CPU 的内部硬件操作如下:

(1) 对 WREG 的内容取补码;

(2) 将补码与立即数 k 相加;

(3) 忽略借位;

(4) 检查 N(负数)标志位,看结果为正还是为负。

```
MOVLW 0x23      ;WREG 23H
SUBLW 0x45      ;WREG = 45H - 23H = 22H
```

```
 45H      0100 0101              0100 0101
-23H      0010 0011 2's comp    +1101 1101
-------                          ----------------
+22H                             0010 0010
Because D7 (the N flag) is 0, the result is
positive.
```

该条指令按下面的规则来设置负数标志位：

	N	
若立即数 k>WREG	0	结果是正数
若立即数 k=WREG	0	结果是零
若立即数 k<WREG	1	结果是负数，用补码表示

例如：

```
    MOVLW 0x98      ;WREG 98H
    SUBLW 0x66      ;WREG = 66H - 98H = CEH

 66H      0110 0110              0110 0110
-98H      1001 1000 2's comp    +0110 1000
------                           ----------
 CEH                             1100 1110
Because D7 (the N flag) is 1, the result is
negative and in 2's comp.
```

SUBWF　　　　　从 fileReg 中减去 WREG

功能：从 fileReg 减去 WREG 的内容（目的地址的内容=fileReg 的内容－WREG 的内容）

语法：SUBWF f,d,a

该条指令用来将 fileReg 的值减去 WREG 的值，并且把结果放入到 WREG(若 d=0)或者 fileReg(若 d=1)中。在执行减法时，CPU 的内部硬件操作如下：

(1) 对 WREG 的内容取补码；

(2) 将补码加到 fileReg 寄存器；

(3) 忽略借位；

(4) 检查 N(负数)标志位，看结果为正还是为负。

例如：

```
MyReg SET 0x20 ;set aside loc 0x20 for MyReg
MOVLW 0x45     ;WREG 45H
MOVWF MyReg    ;MYReg = 45H
MOVLW 0x23     ;WREG = 23H
SUBWF MyReg,F  ;MyReg = 45H - 23H = 22H
 45H      0100 0101              0100 0101
-23H      0010 0011 2's comp    +1101 1101
-------                          ----------------
+22H                             0010 0010
Because D7 (the N flag) is 0, the result is
positive.
```

该条指令按下面的规则来设置负数标志位：

	N	
若 fileReg>WREG	0	结果是正数
若 fileReg=WREG	0	结果是零
若 fileReg<WREG	1	结果是负数，用补码表示

SUBWFB　　　　从 fileReg 中减去 WREG 和借位

功能：执行 fileReg－WREG－＃borrow 操作；＃borrow 是借位

语法：SUBWFB f, d, a

该指令用来从 fileReg 中减去 WREG 的内容和借位，并且把结果存放入 WREG（若 d=0）或者 fileReg（若 d=1）中。在执行减法时，CPU 的内部硬件操作如下：

(1) 对 WREG 的内容取补码；

(2) 将补码加到 fileReg 中；

(3) 减去借位；

(4) 忽略借位；

(5) 检查 N（负数）标志位，看结果为正还是为负。

例如：

```
    MyReg SET 0x20 ;set aside loc 0x20 for MyReg
    BSF     STATUS,C  ;C = 1
    MOVLW  0x45      ;WREG 45H
    MOVWF  MyReg     ;MYReg = 45H
    MOVLW  0x23      ;WREG = 23H
    SUBWFB MyReg,F ;MyReg = 45H - 23H - 0 = 22H

 45H      0100 0101              0100 0101
-23H      0010 0011 2's comp   +1101 1101
               Inverted carry +         0
-----                          -----------
+22H                              0010 0010
Because  D7  (the  N  flag)  is  0,  the  result  is
positive.
```

715

该条指令按下面的规则来设置负数标志位：

	N	
若 fileReg>(WREG ＋ ＃C)	0	结果是正数
若 fileReg=(WREG＋ ＃C)	0	结果是零
若 fileReg<(WREG＋ ＃C)	1	结果是负数，用补码表示

SWAPF　　　　交换 fileReg 的高低半字节

功能：对 fileReg 的内容作高低字节交换

语法：SWAPF f, d, a

SWAPF 指令用于交换 fileReg 中内容的低半字节和高半字节。所得的结果可以存放在 WREG（若 d=0）或者 fileReg（若 d=1）中。

例如:

```
MyReg SET 0X20 ;set aside loc 20H for MyReg
MOVLW 0x59H  ;W = 59H (0101 1001 in binary)
MOVWF MyReg    ;MyReg = 59H (0101 1001)
SWAPF MyReg,F  ;MyReg = 95H (1001 0101)
```

TBLRD　　　　表格读取

功能:从 ROM 中读取 1B 数据到 TABLAT 寄存器

语法:TBLRD*

TBLRD*+

TBLRD*-

TBLRD+*

该指令把程序(代码)ROM 里的 1B 数据送入 TableLatch(TABLAT)寄存器。它允许把程序代码空间中的一串数据(如查表项)读入 CPU。在程序空间(片上 ROM)中操作数的地址由 TBLPTR 寄存器指定。表 A-6 给出了 TBLRD 指令的自增量特性。

表 A-6　PIC18 的读表指令

指　令	功　能
TBLRD*	读表读表后,TBLPTR 保持不变
TBLRD*+	读表,然后加 1(读表并且 TBLPTR 加 1)
TBLRD*-	读表,然后减 1(读表并且 TBLPTR 减 1)
TBLRD+*	读表,之前加 1(TBLPTR 加 1,读表)

注意:从代码空间读入 TABLAT 寄存器的字节数据所在的地址由 TBLPTR 寄存器指定。

例如:假设一个 ASCII 字符串是存储在片上 ROM 程序存储器中,起始地址为 500H。编制程序,把每个字符读入 CPU,再送入到 PORTB 中。

716

```
ORG   0000H            ;burn into ROM starting at 0
      MOVLW LOW(MESSAGE)   ;WREG =`00 low-byte addr
      MOVWF TBLPTRL        ;look-up table low-byte addr
      MOVLW HIGH(MESSAGE)  ;WREG = 05 = high-byte addr
      MOVWF TBLPTRH        ;look-up table high-byte addr
      CLRF  TBLPTRU        ;clear upper 5 bits

B8    TBLRD*+      ;read the table,then increment TBLPTR
      MOVF  TABLAT,W       ;copy to WREG (Z = 1 if null)
      BZ    EXIT           ;exit if end of string
      MOVWF PORTB          ;copy WREG to PORTB
      BRA   B8
EXIT GOTO EXIT
;---------------------message
      ORG 0x500        ;data burned starting at 0x500
          ORG  0x500
MESSAGE   DB   "The earth is but one country and "
          DB   "mankind its citizens","Baha'u'llah",0
          END
```

在上面的程序中,TBLPTR 里存放了对象字节的地址。在执行完 TBLRD*+指令后,寄

存器 TABLAT 里就有字符。注意，TBLPTR 会自动加 1，以指向 MRESSAGE 表的下一个字符。

TBLWT　　　表格写入

功能：向 Flash 写入数据块

语法：TBLWD*

　　TBLWD*＋

　　TBLWD*－

　　TBLWD+*

当片上程序 ROM 是闪存型的时候，这条指令向程序(代码)空间写入数据块。对象的地址由 TBLPTR 确定。关于如何使用 TBLWT 指令向 Flash Rom 写入数据，请参阅 14.3 节。

TSTFSZ　　　测试 fileReg，若为零则跳过

功能：测试 fileReg 的内容是否为 0，如果是 0 就跳过下一条指令

语法：TSTFSZ f, a

该指令用来测试 fileReg 的内容是否为 0，如果是 0 就跳过下一条指令。

例如：连续地测试 PORTB 是否为 0。

717

```
SETF TRISB  ;make PORTB an input
CLRF TRISD  ;make PORTD an output
BACK TSTFSZ PORTB
     BRA    BACK
     MOVFF  PORTB,PORTD
```

又例如：翻转 PORTB 250 次。

```
COUNTER   SET  0X40 ;loC 40H for COUNTER
     SETF TRISB     ;PORTB as output
     MOVLW D'250'   ;WREG = 250
     MOVWF COUNTER  ;COUNTER = 250
     MOVLW 0x55     ;WREG = 55H
     MOVWF PORTB
BACK COMF PORTB,F   ;toggle PORTB
     DECF COUNTER,F ;decrement COUNTER
     TSTFSZ COUNTER ;test counter for 0
     BRA  BACK      ;keep doing it
     ......
```

XORLW　　　将立即数与 WREG 的内容做逻辑异或运算

功能：将立即数 k 和 WREG 的内容做逻辑异或操作

语法：XORLW k

该条指令用来执行立即数 k 和 WREG 操作数的逻辑异或操作，按位地进行，结果放在 WREG 中。

例如：

```
MOVLW 0x39   ;WREG = 39H
XORLW 0x09   ;WREG = 39H ORed with 09
```

```
                   ;now, WREG = 30H
39H    0011 1001
09H    0000 1001
30H    0011 0000
```

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

又例如:

```
MOVLW 0x32      ;WREG = 32H
XORLW 0x50      ;(now, WREG = 62H)

32H    0011 0010
50H    0101 0000
62H    0110 0010
```

XORWF　　将 WREG 的内容与 fileReg 的内容做异或运算

功能:将 fileReg 的内容和 WREG 寄存器的内容做逻辑异或运算

语法:XORWR f,d,a

该条指令对操作数执行逻辑异或运算,逐位地进行,结果存放在目的地址。目的地址可以是 WREG(若 d=0),或者是 fileReg(若 d=1)。 [718]

例如:

```
MyReg SET 0x20 ;set aside loc 20h for MyReg
MOVLW 0x39     ;WREG = 39H
MOVWF MyReg    ;MyReg = 39H
MOVLW 0x09     ;WREG = 09H
XORWF MyReg,F  ;MyReg = 39H ORed with 09
               ;MyReg = 30H

39H    0011 1001
09H    0000 1001
30H    0011 0000
```

又例如:

```
MyReg SET 0x15 ;set aside loc 15 for MyReg
MOVLW 0x32     ;WREG = 32H
MOVWF MyReg    ;MyReg = 32H
MOVLW 0x50     ;WREG = 50H
XORWF MyReg,F. ;now W = 62H

32H    0011 0010
50H    0101 0000
62H    0110 0010
```

也可以把结果放入 WREG 寄存器,请看例子。

例如:

```
MyReg SET 0x15 ;set aside loc 15 for MyReg
MOVLW 0x44     ;WREG = 44H
MOVWF MyReg    ;MyReg = 44H
MOVLW 0x67     ;WREG = 67H
XORWF MyReg    ;now W = 23H, and MyReg = 44H

44H   0100 0100
67H   0110 0111
23H   0010 0011
```

[719]

附录 B 绕线基础知识

概述

本附录将介绍绕线的一些基础知识。

绕线基础

注意，在本附录指南的学习中，可能需要下面的工具和材料：

绕线工具(Radio Shack，产品型号：276-1570)

用于绕线的 30 口径(30-AWG)的接线

(在此，感谢 Shannon Looper 和 Greg Boyle 对本节编写的协助。)

下面是对绕线基础知识的介绍。

(1) 目前有很多不同种类的绕线工具可用。其中最好的就是 Radio Shack，它的价格还不到 10 美元。Radio Shack 工具的型号是 276-1570。该工具集合绕线和解绕功能于同一终端，而且还带有一个独立的剥线器。相比那些把所有功能都集中在一个二端轴的工具，这种安排的工具更容易使用。还有一些是带有绕线焊接枪的，当然是很贵的。

(2) 用于绕线的接线可以是预先切割好的不同长度的，也可以是线轴上的一大捆。预先切割好的接线通常会更贵，而且会受到所购买的接线长度的限制。一大捆的接线可以让你根据需要切成任意长度，也就是每条线都是可以定制的。

(3) 绕线板的种类也是多种多样的。通常称为穿孔板或者绕线板。这些电路板在很多电子商店都有出售(如 Radio Shack)。最好的绕线板应该是在底部有电镀孔的。这些绕线板更好用，因为插槽和引脚可以被焊接到板上，让电路的机械结构更稳固。

(4) 选择一块足够大的电路板，让它能容纳你设计的所有元器件，那样线路就不会太混乱。如果你在将来还希望扩充设计的话，那么就需要保证原来的板有足够的空间放置整个电路。同时，如果可能的话，电路板上的 IC 的布局也应该像电路图那样让信号从左往右传。

(5) 让接线尽量简单，而且不要让引脚受压，并在电路板的每个角上都装上支架。当电路板水平安装时，你也可以在电路板的上面加上支架来提高机械稳固性。

(6) 对于电源接头，可使用标准的接线柱。焊接一些单接线的引脚到每一个通电的接线柱，保证电路连接(电路中每片 IC 至少有一个这样的引脚)。

(7) 为了减少电源的问题，每片 IC 都必须独立地连接到电路板的主电源。如果你的电路板没有内部电源总线，那就采用独立的电源和地线连接每一片 IC 和主电源。也就是说，不要用菊花链(芯片到芯片的连接称为菊花链)的连接，因为那样每条线路需要更多连线而且电阻值更大。如图 B-1 所示。然而，菊花链对于其他连接(如数据、地址还有控制总线)是非常有用的。

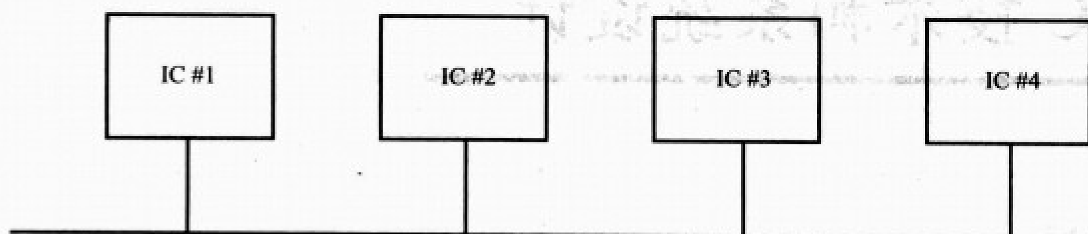


图 B-1 菊花链连接(不推荐用于电源线连接)

(8) 必须使用绕线插座。这种插座有长方形插脚，在绕线时可直接切入到线路中。

722

(9) 绕线对圆插脚不起作用。如果需要绕接到圆形引脚的元件(如电容)，必须对这些引脚进行焊接连接。连接单个元件的最好方法是在电路板上安装独立的绕线引脚，然后把元件焊接到引脚上。另外一个办法就是，使用空 IC 插座来放置小元件(如电阻)，把它们一起绕接到插座上。

(10) 必须将接线切成 1 英寸长短的。这使得每个连接能有 7 到 10 圈。第 1 圈或者第 1 圈半必须是绝缘的，这样可以阻止接线和其他引脚的接触。实现的方法就是在连接之前将接线插入到它能穿越的地方。

(11) 尽量使绕线的长度最短。这可以防止电路的杂乱。应尽可能地保持线路的整齐，尽可能使用颜色编码。红色作为 V_{cc}，而黑色作为地线。同时用不同的线来连接数据、地址以及控制信号。使用这个方法可以更容易地发现错误。

(12) 标准的操作是先连上所有的电源线，确保它们的连通性。这样可以减少后面可能出现的麻烦。

(13) 在电路板的底部标注引脚的方向也是一个好办法。对此，可以在塑料模版上对引脚印好编号，也可以用纸来写。在看电路板底部的标注时忘记倒转引脚顺序是绕线时常见的错误。

(14) 为了保护电路免受损坏，应加上一个二极管(如 IN5338)来应对电源的反偏电压。如果电源突然反向，二极管正向偏压，就会短路，从而阻止反向电压进入电路。

(15) 对于数字电路，一个问题是电源提供的电流。为了滤除电源的噪声，可将一个 100 μF的电解电容和一个 0.1 μF 的单片电容在电路板的电源连接入口处并联连接到 V_{cc}和地之间。这两个电容可以滤掉高频和低频噪声。如果不用两个并联的电容，也可以使用一个 20～100 μF的钽电容。记住，长的引脚为正极性。

(16) 为了滤掉瞬态电流，应在每片 IC 上加一个 0.1 μF 单片电容。将 0.1 μF 单片电容连接在每片 IC 的 V_{cc}和地之间。应确保引脚尽可能地短。

723

附录 C IC 技术和系统设计

概述

本附录综述了 IC 技术和 PIC18 接口技术。另外，还把基于微控制器的系统作为一个整体，讨论系统设计的常见问题。

C.1 节综述了 IC 技术。C.2 节介绍了 PIC18 的 I/O 端口的内部细节。C.3 节讨论了系统设计的问题。

C.1 IC 技术概述

本节将介绍 IC 技术并讨论一些高级逻辑器件的主要发展。因为这只是一个概述，所以假定读者已经对基础数字电路教材中的逻辑器件有相当的熟悉。

C.1.1 晶体管

晶体管是 1947 年由贝尔实验室的 3 位科学家发明的。在 20 世纪 50 年代，晶体管取代了许多电子系统（包括计算机）中的真空管。不过，直到 1959 年，第一片集成电路才由德州仪器公司的 Jack Kilby 成功地制造和测试。在 IC 发明以前，晶体管以及其他独立元件（如电容和电阻）都常用在计算机设计中。早期的晶体管由锗制造，后来被硅制造取代。这是因为在大量电流流过由锗制造的晶体管时温度有轻微的升高。对于半导体来说，锗晶体管的沟道比硅晶体管的小，因此，即使在温度上升很轻微的时候，也有大量电子从电子区流向导通区。从 20 世纪 50 年代末到 20 世纪 70 年代初，基于硅的 IC 开始广泛地应用在大型机和小型计算机上。晶体管和 IC 的制造最初是基于 P 型材料的。后来，因为电子的速度比空穴更快（大约是 2.5 倍），于是 N 型设备就取代了 P 型设备。到了 20 世纪 70 年代中期，NPN 和 NMOS 晶体管已经在电子工业的各个领域（包括微处理器和计算机）取代了速度较慢的 PNP 和 PMOS 晶体管。自 20 世纪 80 年代早期以来，CMOS 成为 IC 设计的主导技术。下面简单介绍 MOS 管和双极型晶体管的区别。如图 C-1 所示。

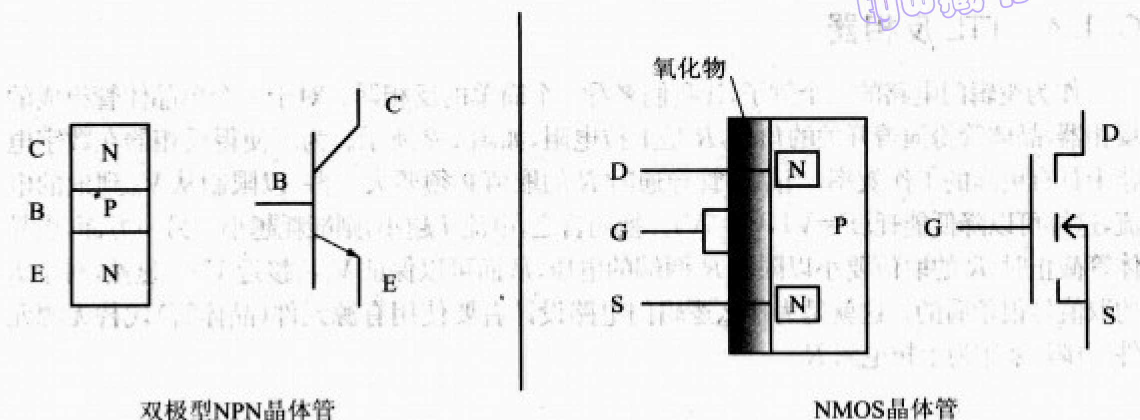


图 C-1 双极型晶体管与 MOS 晶体管的对比

726

C.1.2 MOS 和双极型晶体管的对比

晶体管有两类:双极型和 MOS(金属氧化物半导体)型。两种都是带有 3 个引脚。双极型晶体管的 3 个引脚分别对应着发射极、基极和集电极,而 MOS 晶体管的 3 个引脚则对应着源极、栅极和漏极。在双极型晶体管里,载流子从发射极流到集电极,基极起电流控制作用。在 MOS 晶体管里,载流子从源极流向漏极,栅极起电压控制作用。在 NPN 双极型晶体管里,电子载流子从发射极出发必须经过两个势垒才能到达集电极(见图 C-1)。一个是发射极和基极间的 NP 结,另一个是基极和集电极之间的 PN 结。基极和集电极间的势垒对于电子来说是最难克服的(因为它反向偏置)而且能量损耗最多。这种缺陷推动了单极型晶体管——MOS 管的诞生。对于 N 沟道 MOS 晶体管,电子从源极流到漏极而无需经过任何势垒。由于没有势垒阻挡,MOS 损耗的能量比双极晶体管要小得多。MOS 的低损耗特性使得成千上万的晶体管可以工作在同一片 IC 上。今天,由于 MOS 技术的进步,即使让 1 千万个晶体管集成到一片 IC 上,也是再平常不过的事情了。要是没有 MOS 晶体管的问世,个人笔记本电脑就不会实现,至少没这么快。20 世纪 60 年代~70 年代采用的双极型晶体管制造的大型机和小型机很庞大,而且需要昂贵的冷却设备和很大的物理空间。MOS 晶体管也有它的缺点:在速度上较双极型晶体管要慢。因为 MOS 需要翻转,输入需要时间来充电以达到门限(阈值)电压,相应地导致了较长的时延。

C.1.3 逻辑器件综述

逻辑器件的判断有几个指标:(1)速度;(2)能耗;(3)抗噪声能力;(4)输入/输出接口的兼容性;(5)价格。理想的性能是高速、低能耗、高的抗噪声能力。较高的抗噪声能力可以避免在开关转换时出现错误的逻辑信号。对于接口逻辑器件,一个输出端口可以驱动的输入端口越多就越好。也就是说,高驱动性能的输出是理想的。鉴于 MOS 和双极型晶体管的输入输出电压是不兼容的,这就必须要考虑一种逻辑器件对另外一种逻辑器件的驱动能力。在价格方面,逻辑器件在推出的早期,价格自然会在相当长的时间里高居不下,但是随着生产和使用量的增加,价格则会持续下降。

C.1.4 TTL反相器

作为逻辑门电路的一个例子，让我们来看一个简单的反相器。对于一个单晶体管组成的反相器，晶体管扮演着开关的角色，*R* 是上拉电阻，如图 C-2 所示。为了使得反相器在数字电路中具有更高的工作效率，当晶体管导通时 *R* 的取值必须要大一些，以限制从 V_{cc} 到地的电流，这样可以降低能耗（$P=VI,V=5V$）。换而言之，电流 *I* 越小，则能耗越小。另一方面，当晶体管截止时，*R* 的取值要小以限制 *R* 两端的电压，从而可以保证 V_{OUT} 接近 V_{cc}。显然，对于 *R* 的取值是很矛盾的。这就是为什么逻辑门电路设计者要使用有源元件（晶体管）代替无源元件（电阻）来作为上拉电阻 *R*。

727

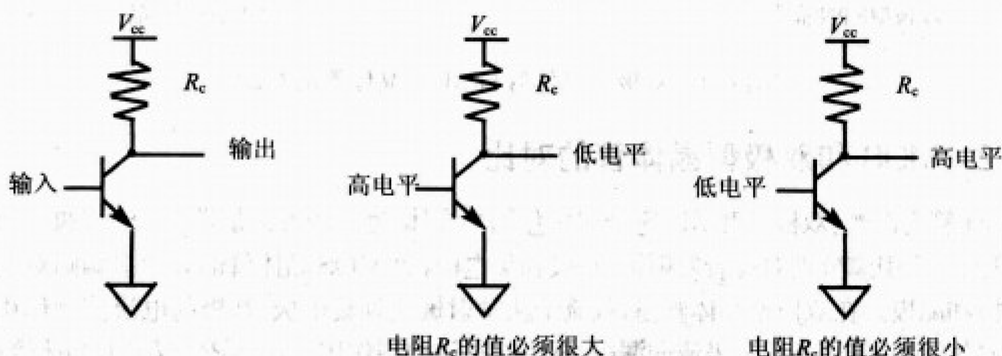
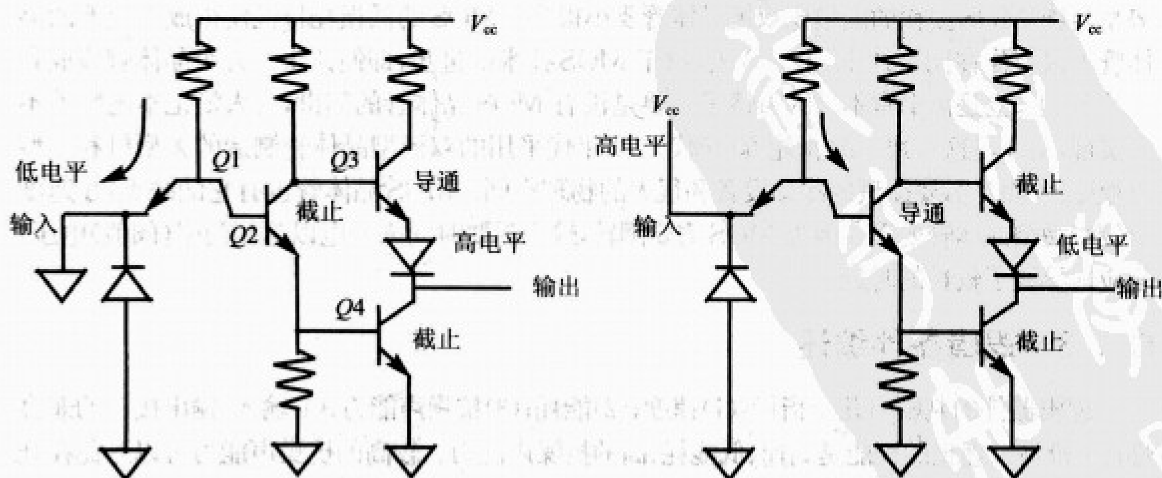


图 C-2　带上拉电阻的单晶体管反相器

一个带推拉式输出的 TTL 反相器如图 C-3 所示的。在图 C-3 中，*Q3* 起到上拉电阻的作用。



728

图 C-3　带推拉式输出的 TTL 反相器

C.1.5 CMOS反相器

对于基于 CMOS 的逻辑门而言，PMOS 和 NMOS 常用来构成 CMOS 反相器，如图 C-4

所示。在CMOS反相器中,当PMOS晶体管截止时,提供一个高阻抗通路,使得漏电流几乎为0(大约10 nA);当PMOS晶体管导通时,则提供从V_{DD}到负载的低阻抗通路。因为空穴的速度比电子要慢,而PMOS晶体管更多地补偿这种不平衡;因此,PMOS晶体管在CMOS门电路里要比NMOS晶体管占用更多的空间。在本节的最后,读者会看到支持外接上拉电阻的集电极开路的门电路,因此,系统设计者可以选择合适阻值的上拉电阻。

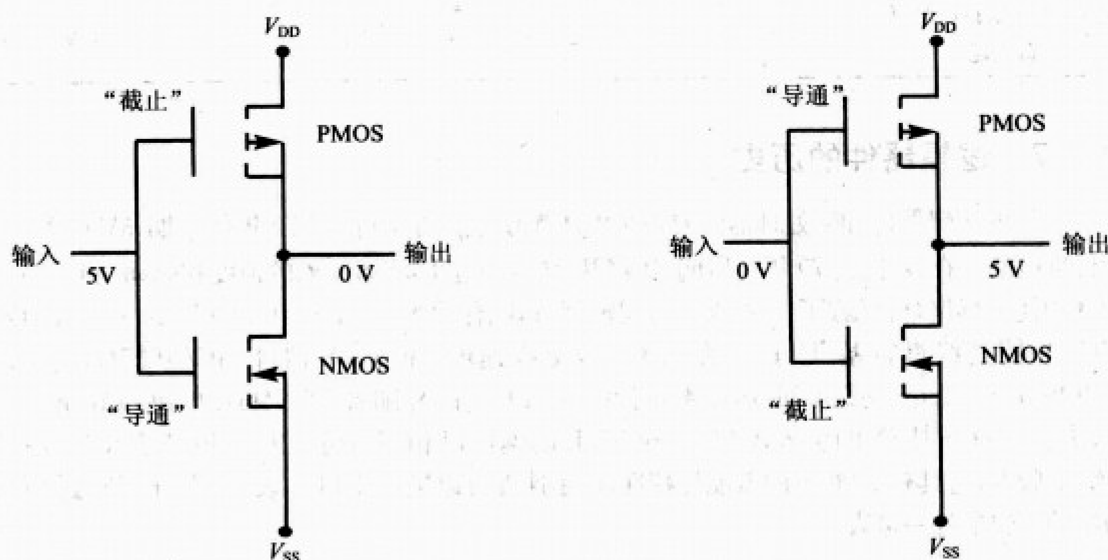


图 C-4 CMOS反相器

C.1.6 部分逻辑器件的输入/输出特性

在1968年,由双极型晶体管制造的第一代逻辑器件投入市场。它们通常被称为标准的TTL(Transistor-Transistor Logic,晶体管-晶体管逻辑)器件。第一批基于MOS的逻辑器件CD4000/74C系列,在1970年投入市场。在20世纪70年代早期的双极型晶体管中引入肖特基二极管,诞生了S系列逻辑器件。肖特基二极管的作用在于阻止集电极进入所谓的深度饱和,从而缩短了TTL的传输时间延迟。表C-1列出了部分逻辑器件的主要特性。在表C-1中可以注意到,当CMOS电路的工作频率提高的时候,能耗也会有所增加。对于基于双极型的TTL逻辑门电路则不会出现这种情况。

表 C-1 部分逻辑器件的主要特性

特　　性	STD TTL	LSTTL	ALSTTL	HCOMS
V_{CC}	5 V	5 V	5 V	5 V
V_{IH}	2.0 V	2.0 V	2.0 V	3.15 V
V_{IL}	0.8 V	0.8 V	0.8 V	1.1 V
V_{OH}	2.4 V	2.7 V	2.7 V	3.7 V
V_{OL}	0.4 V	0.5 V	0.4 V	0.4 V
I_{IL}	−1.6 mA	−0.36 mA	−0.2 mA	−1 μA
I_{IH}	40 μA	20 μA	20 μA	1 μA

（续）

特　　性	STD TTL	LSTTL	ALSTTL	HCOMS
I_{OL}	16 mA	8 mA	4 mA	4 mA
I_{OH}	−400 μA	−400 μA	−400 μA	4 mA
传输时延	10 ns	9.5 ns	4 ns	9 ns
静态能耗(f=0)	10 mW	2 mW	1 mW	0.0025 nW
在 f=100 kHz 时的动态能耗	10 mW	2 mW	1 mW	0.17 mW

729

C.1.7　逻辑器件的历史

早期的逻辑器件和微处理器需要正负电压源供电。在 20 世纪 70 年代中期，5V 的 V_{cc}成为标准电压。在 20 世纪 70 年代后期，随着 IC 技术的进步，将 S 系列的速度和驱动同 LS 系列的低功耗进行整合，形成了一种新型的逻辑器件，叫作 FAST（Fairchild advanced schottky TTL，快捷高级肖特基 TTL）。在 1985 年，更高速的 HCMOS 器件 AC/ACT（advanced CMOS technology，高级 CMOS 技术）问世。在 1986 年，伴随着 FCT（fast CMOS technology，快速 CMOS 技术）的引入，CMOS 和 TTL 的速度已相当接近。因为 FCT 是 CMOS 的 FAST 版本，它具有 CMOS 的低能耗特性，但是速度可以接近 TTL。表 C-2 给出了直到 FCT 版本的逻辑器件一览表。

表 C-2　逻辑器件一览表

产　　品	问世时间	速度(ns)	静态电流(mA)	高/低电平驱动电流(mA)
Std TTL	1968	40	30	−2/32
CD4K/74C	1970	70	0.3	−0.48/6.4
LS/S	1971	18	54	−15/24
HC/HCT	1977	25	0.08	−6/−6
FAST	1978	6.5	90	−15/64
AS	1980	6.2	90	−15/64
ALS	1980	10	27	−15/64
AC/ACT	1985	10	0.08	−24/24
FCT	1986	6.5	1.5	−15/64

转载已获得 *Electronic Design Magazine*（C. 1991）的许可。

C.1.8　逻辑器件的最新进展

随着高性能微处理器的速度达到 25 MHz，CPU 的周期大大缩短，传输延迟时间也大为降低。设计者通常分配给传输延迟的时间不超过 CPU 周期的 25%。遵循这一点，在增加系统工作频率时，逻辑器件花在数据和地址传输路径上的延迟时间会有相应的减少。近年来，针对这种需求很多半导体生产商推出了高速、低噪声和高 I/O 驱动能力的逻辑器件。表 C-3 列出了近年生产的高性能逻辑器件。ACQ/ACTQ 是第二代高级 CMOS(ACMOS)，具有更低的噪声。ACQ 支持 CMOS 电平输入，而 ACTQ 则需要 TTL 电平输入。FCTx 和 FCTx-T 是第二代 FCT，具有更高的速度。FCTx 和 FCTx-T 中的 x 表示不同的速度，例如用 A 代表

低速，C代表高速。对于精通 FAST 逻辑器件的设计者来说，FASTr 是较为理想的选择，因为它的速度比 FAST 更高，并且相比 FAST，它具有更高的驱动性能（I_{OL}，I_{OH}）而且噪声更小。在本书编写的时候，FASTr 是市场上最快的逻辑器件（5 V V_{cc}），仅次于 ECL 和镓砷化物逻辑门电路，但是同其他逻辑器件一样，它的能耗也较高。如表 C-3 所示。将高速的双极型 TTL 和低能耗的 CMOS 相结合，形成了新的逻辑器件 BICMOS。尽管 BICMOS 代表着未来 IC 设计的方向，可是目前在生产制造中所增加的工艺使得 BICMOS 的价格非常昂贵，然而在某些方面设计人员是没有其他选择的。例如，英特尔公司的奔腾处理器就是 BICMOS 产品，它必须使用高速双极型晶体管来提高内部的性能。表 C-3 列出了高级逻辑器件的通用特性。这里，x 代表不同的速度，例如 A 代表该系列速度最慢的逻辑器件，而 C 代表该系列速度最快的逻辑器件。表 C-3 是关于 74244 缓冲器的数据。

730

从 20 世纪 70 年代后期开始，+5 V 的电源成为所有微控制器和微处理器的标准。为了减少能耗，3.3 V 的 V_{cc} 现在也被很多设计者所推崇。将 V_{cc} 降低到 3.3 V 有两个主要的好处：(1)降低了能耗，可以延长采用电池供电的系统的电池寿命；(2)走线的宽度（设计规则）进一步降低到亚微米级。这条走线宽度的降低使得在给定的硬膜上可以容纳更多的晶体管。随着工艺过程的改进，走线宽度降至亚微米级，晶体管的密度接近 10 亿。

表 C-3 高级逻辑器件的通用性能

产品系列	问世时间	供应商数量	技术基础	I/O 电平	速度(ns)	静态电流	I_{OH}/I_{OL}
ACQ	1989	2	CMOS	CMOS/CMOS	6.0	80 μA	−24/24 mA
ACTQ	1989	2	CMOS	TTL/CMOS	7.5	80 μA	−24/24 mA
FCTx	1987	3	CMOS	TTL/CMOS	4.1～4.8	1.5 mA	−15/64 mA
FCTxT	1990	2	CMOS	TTL/TTL	4.1～4.8	1.5 mA	−15/64 mA
FASTr	1990	1	BICMOS	TTL/TTL	3.9	50 mA	−15/64 mA
BCT	1987	2	BICMOS	TTL/TTL	5.5	10 mA	−15/64 mA

转载已获得 *Electronic Design Magazine*（C. 1991）的许可。

C.1.9 集电极开路和漏极开路的门电路

为了让更多的输出可以连在一起，可以使用集电极开路的逻辑电路。在这种情况下，外接的上拉电阻就充当负载，如图 C-5 和图 C-6 所示。

731

C.2 PIC18 的 I/O 端口结构和接口

要将 PIC18 微控制器和其他 IC 芯片或者设备连接，扇出能力是最重要的问题。为了理解 PIC18 的扇出能力，首先要理解 PIC18 的端口结构。本节将详细讨论 PIC18 的端口结构与扇出能力。理解 PIC18 的端口结构是非常重要的，以避免在将外部设备连接到 PIC18 时损坏器件。

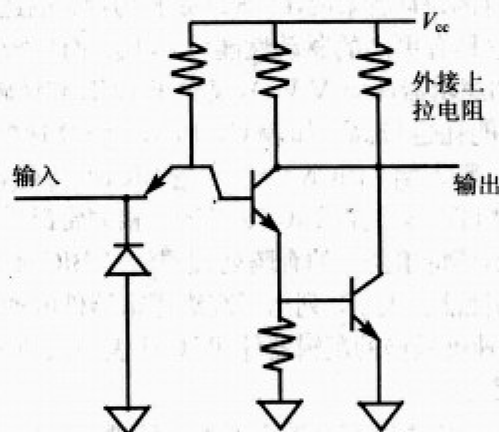


图 C-5 集电极开路



图 C-6 开漏极

C.2.1 IC 扇出能力

当把 IC 芯片连接到一起的时候，需要确定一个输出引脚能驱动多少个输入引脚。这个问题非常重要，涉及 IC 扇出能力的概念。IC 扇出能力必须分别考虑输出为逻辑 0 和逻辑 1 的情况。如例 C-1 所示。逻辑低电平的扇出能力和逻辑高电平的扇出能力分别定义如下：

$$扇出能力(低电平)=\frac{I_{OL}}{I_{IL}} \qquad 扇出能力(高电平)=\frac{I_{OH}}{I_{IH}}$$

在上面的两个值中，较低的数可用来确保适当的噪声裕度。图 C-7 画出了 IC 芯片连接在一起时的灌电流和拉电流情况。

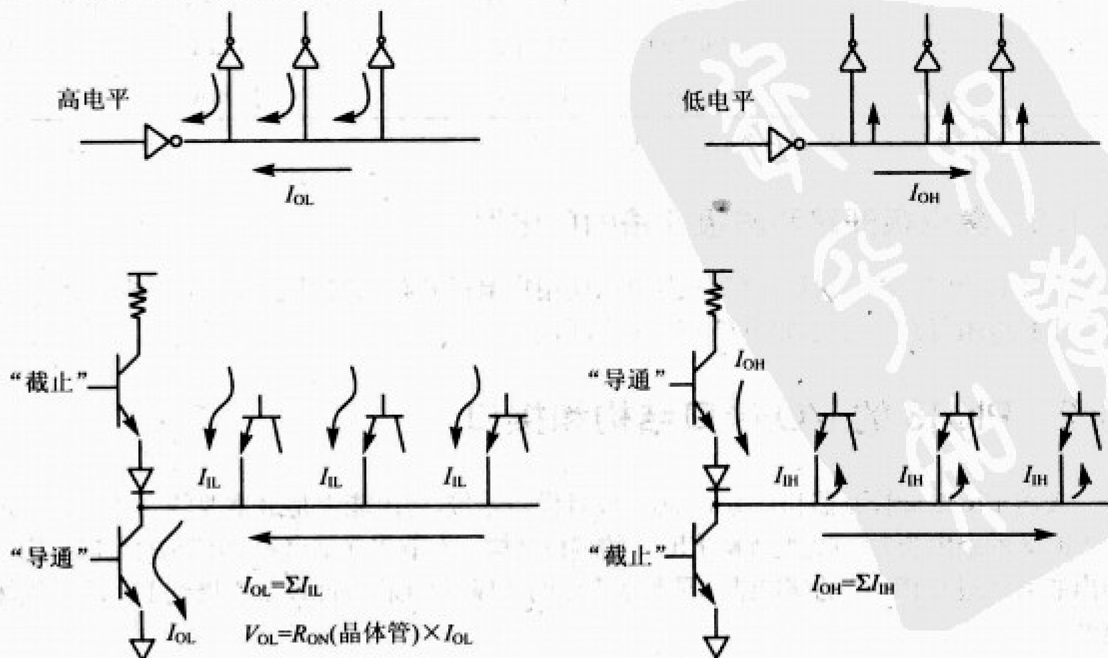


图 C-7 TTL 中的灌电流和拉电流

注意,在图 C-7 中,当连接到同一输出端的输入引脚数目增加时,I_{OL}将上升,从而使得 V_{OL} 也上升。如果输入引脚数目继续增加,V_{OL}的上升将会使得噪声裕度变小,于是会出现因微弱噪声引起的错误逻辑。

例 C-1 试确定下面的 LS 逻辑器件可以驱动多少单位负载(UL)。

解:将单位负载定义为 $I_{IL}=1.6\ \text{mA}$, $I_{IH}=40\ \mu\text{A}$。查表 C-1 可知,LS 器件的 $I_{OH}=400\ \mu\text{A}$, $I_{OL}=8\ \text{mA}$。因此,可以得出

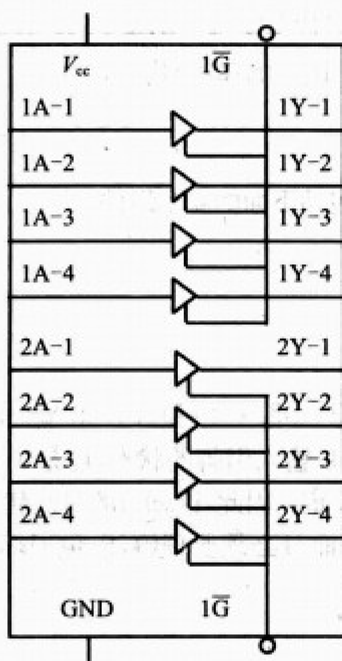
$$扇出能力(低电平)=\frac{I_{OL}}{I_{IL}}=\frac{8\ \text{mA}}{1.6\ \text{mA}}=5$$

$$扇出能力(高电平)=\frac{I_{OH}}{I_{IH}}=\frac{400\ \mu\text{A}}{40\ \mu\text{A}}=10$$

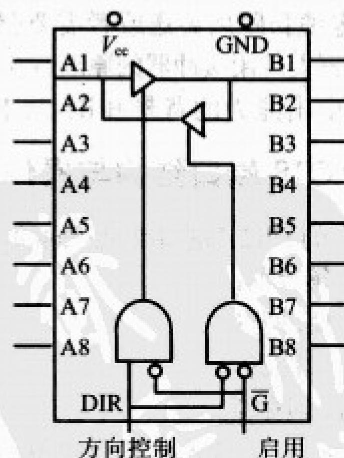
因此,LS 逻辑器件的扇出能力是 5。换而言之,LS 输出最多能连接 5 个单位负载。

C.2.2 74LS244 和 74LS245 缓冲器/驱动器

当接收器的电流需求大于驱动器提供的能力时,必须使用缓冲器/驱动器,如 74LS244 和 74LS245。图 C-8 给出了 74LS244 和 74LS245 的内部逻辑门电路。74LS245 用于双向的数据总线,而 74LS244 用于单向的数据总线。



(a) 74LS244 8 位缓冲器(转载经由德州仪器公司许可,德州仪器版权保护,1988)



功能表

启用 $\overline{G}$	方向控制 DIR	操作
L	L	数据 *B* 到总线 *A*
L	H	数据 *A* 到总线 *B*
H	X	隔离

(b) 74LS245 双向缓冲器(转载经由德州仪器公司许可,德州仪器版权保护,1988)

图 C-8

C.2.3 三态缓冲器

注意，74LS244 是一个简单的 8 位三态缓冲器。如图 C-9 所示，三态缓冲器包括一个输入端、一个输出端和一个使能控制端。当使能端有效时，输入端的数据就传送到输出端。使能端可以是低电平有效也可以是高电平有效。注意，74LS244 的使能端是低电平有效，而图 C-9 中的使能端引脚是高电平有效。

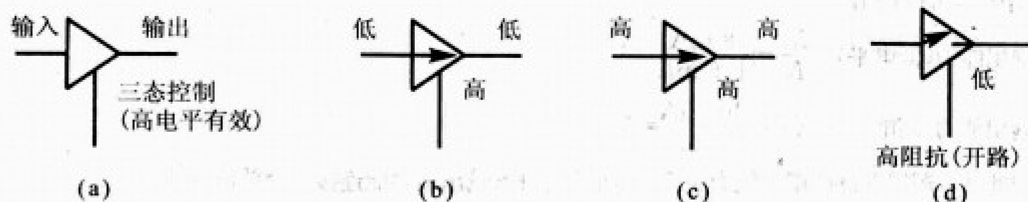


图 C-9 三态缓冲器

C.2.4 74LS245 和 74LS244 的扇出能力

要注意，74LS245 和 74LS244 能提供比其他 LS 门电路更大的灌电流和拉电流，如表 C-4 所示。这就是在信号经由电缆长距离传送或者需要驱动多个输入时，要在驱动器上使用缓冲器的原因所在。

表 C-4 缓冲器/驱动器的电气指标

	I_{OH}(mA)	I_{OL}(mA)
74LS244	3	12
74LS245	3	12

在了解扇出能力的背景知识后，下面继续讨论 PIC18 的端口结构。

C.2.5 PIC18 端口结构与操作

因为所有的 PIC18 端口都是双向的，所以它们的结构包括下面的 4 个组件：

(1) 数据锁存器；

(2) 输出驱动器；

(3) 输入缓冲器；

(4) TRIS 锁存器。

图 C-10 画出了一个端口的结构和它的 4 个组件。注意，在图 C-10 里，PIC18 端口既有锁存器也有缓冲器。现在的问题是，在读端口的时候，读到的是输入引脚的状态，还是锁存器的状态呢？这是一个非常重要的问题，答案由使用的指令来决定。因此，读端口的时候就有两种可能：(1)读输入引脚；(2)读锁存器。上面的区别非常重要而且必须要理解，以免损坏 PIC18 的端口。下面分别讨论这两种情况。

734

C.2.6 当 TRIS＝1 时，读引脚(输入)

正如第 4 章所说，要将 PIC18 端口的某一位用作输出，首先要向 TRIS 的相应位写 1 (逻辑高电平)。要明白其中的奥妙，请看下面的事件顺序。

(1) 如图 C-10 所示，将 1 写入 TRIS 锁存器以后，Q 位变为“高”电平。因此，Q=1，$\overline{Q}$=0。因为 Q=1，所以晶体管 P 截止。

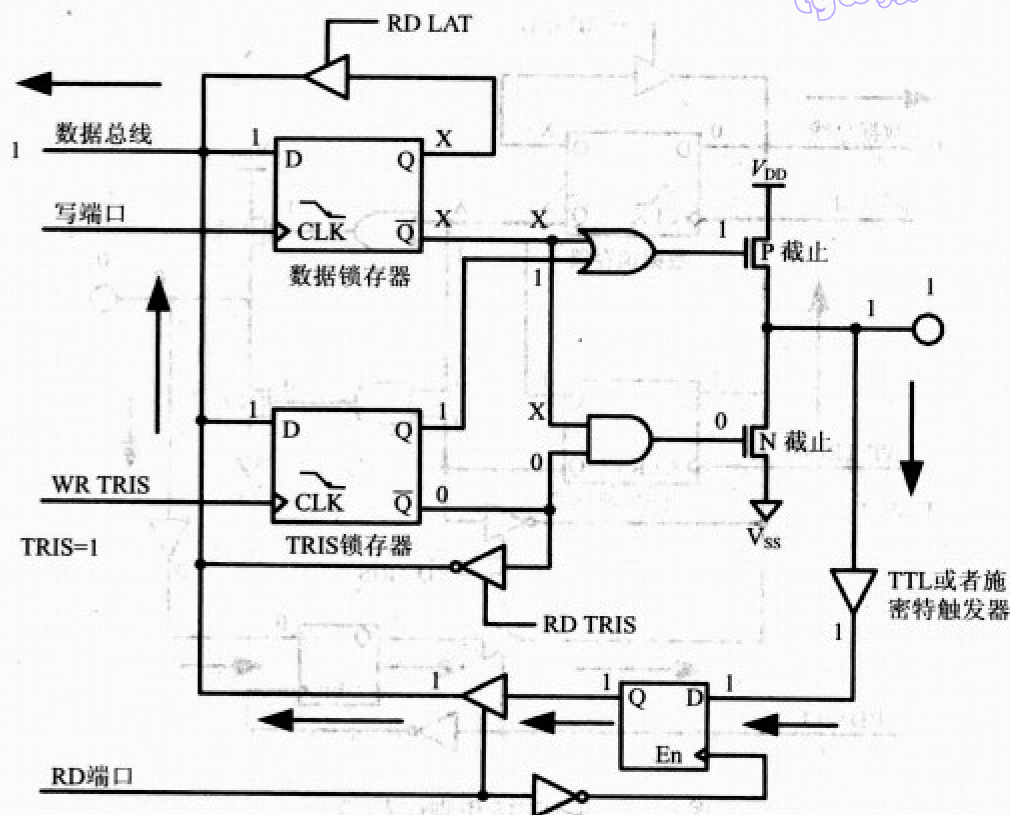


图 C-10 从 PIC18 引脚输入(读)1

(2) 由于 $\overline{Q}$=0,且 $\overline{Q}$ 是连接到 N 晶体管的,所以晶体管 N 截止。

(3) 当两个晶体管都截止时,它们切断了连接到输入引脚的任何信号到地和 V_{CC} 的通路,输入信号将被送入缓冲器。

(4) 当用 MOVFW PORTG 这样的指令读输入端口时,实际上是读引脚上的数据。换而言之,它把外部引脚状态读入到 CPU。这条指令用来读缓冲器的引脚,将引脚上数据传送到 CPU 内部总线上。图 C-10 和图 C-11 分别描述输入信号为高低电平的情况。

C.2.7 当 TRIS=0 时,写引脚(输出)

上面讨论了为什么必须把"高"电平写入端口的 TRIS 位来把它设置为输入端。如果把"0"写入到输入端的 TRIS 会是什么情况呢? 从图 C-12 可以看到,当 TRIS=0 时,如果把 0 写入数据锁存器,那么 Q=0,$\overline{Q}$=1。因为 $\overline{Q}$=1,所以晶体管 N 导通,晶体管 P 截止。如果晶体管 N 导通,那么它为输入引脚提供了接地的通路。因此,所有读输入引脚都会读到"低"电平的接地信号。图 C-13 给出了当 TRIS=0 时把"高"电平写入输出端(数据锁存器)将会发生的情况。把 1 写入数据锁存器,使得 $\overline{Q}$=0。结果是,晶体管 P 导通而晶体管 N 截止,输出引脚为 1。因此,任何的读输入引脚都会读得"高"电平信号。

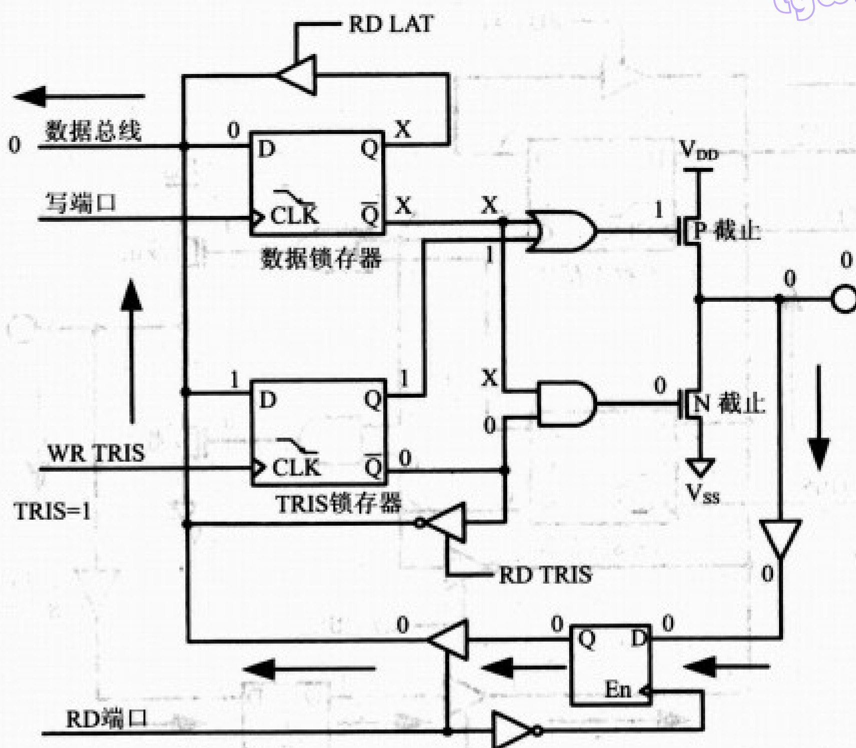


图 C-11　从 PIC18 引脚输入(读)0

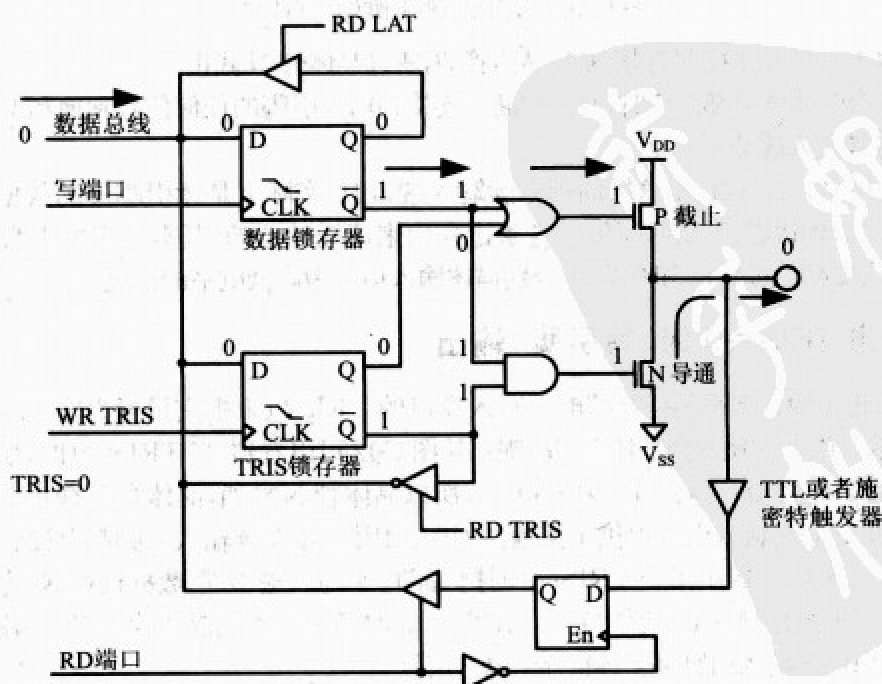


图 C-12　向 PIC18 引脚输出(写)0

C.2.8　避免损坏端口

下面的方法可以用来提醒设计人员避免损坏 PIC18 的端口。

(1) 在 V_{cc} 通路加上一个 10 kΩ 的电阻来限制电流。

(2) 在将输入开关连接到 PIC18 的引脚之前，先连接到三态缓冲器。

上面的两个设计技巧非常重要，而且必须要加以重视。因为很多人在损坏了端口以后都不知道是怎么发生的。当要读取输入引脚状态时，必须要使用正确的指令。表 C-5 列出了读端口时的读输入引脚状态的指令。

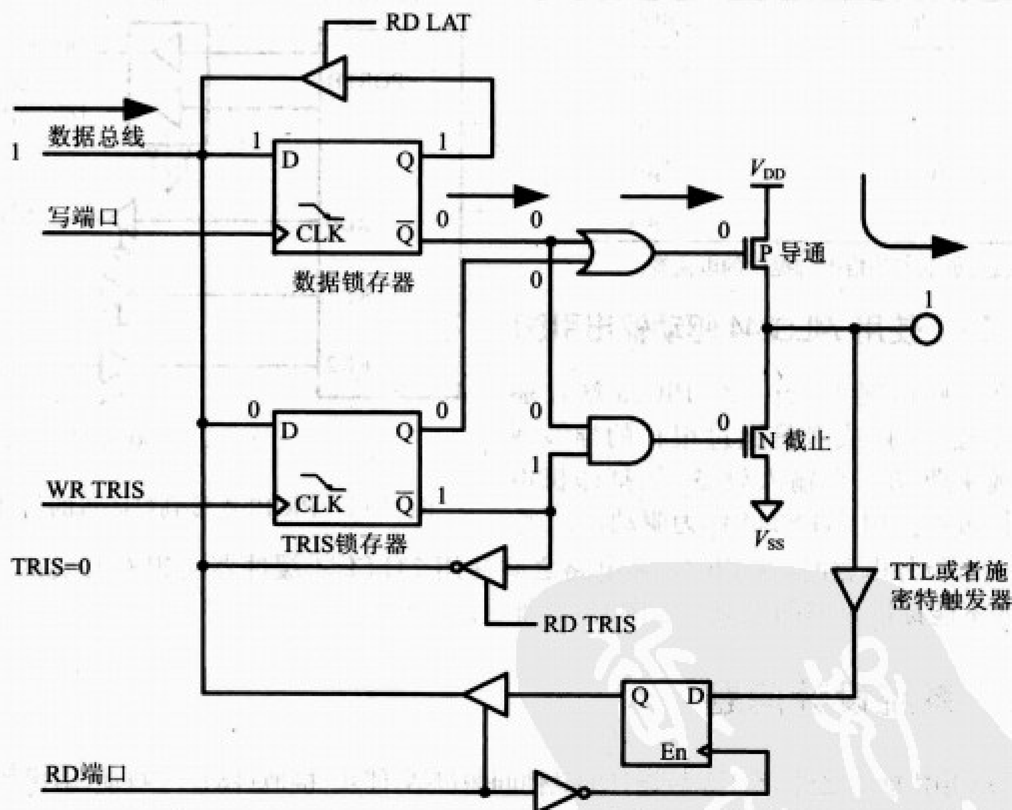


图 C-13　向 PIC18 引脚输出(写)1

736～737

表 C-5　一些读取输入端口状态的指令

助　记　符	例　　子
MOVFW PORTx	MOVFW PORTB
TSTFSZ f	TSTFSZ PORTC
BTFSS f,b	BTFSS PORTD,0
BTFSC f,b	BTFSC PORTB,7
CPFSEQ f	CPFSEQ PORTB

C.2.9　PIC18 端口的扇出能力

在熟悉了 PIC18 端口的结构后，接下来讨论 PIC18 微控制器的扇出能力。早期的芯片是基于 NMOS 的 IC 技术实现的，现在的 PIC18 微控制器全部都是基于 CMOS 技术的。注意，尽管 PIC18 微控制器的核心是 CMOS，但它的引脚驱动电路是 TTL 兼容的，即 PIC18 是具有 TTL 兼容引脚的基于 CMOS 的产品。PIC18 的所有端口都有相同的 I/O 结构，因此扇出能力是相同的。表 C-6 列出了 PIC18F458 端口的 I/O 特性。

表 C-6　PIC18 端口的扇出能力

引脚	扇出
I_{OL}	8.5mA
I_{OH}	−3mA
I_{IL}	1μA
I_{IH}	1μA

注意：负电流是指引脚提供的电流源。

C.2.10　使用 74LS244 驱动输出引脚

在一些情况下，当一个 PIC18 端口驱动多个输入时，或者是通过很长的导线或者电缆来驱动一个输入设备（如打印机电缆）时，可以使用 74LS244 作为驱动器。当驱动一个板外电路时，在 PIC18 和电路之间使用 74LS244 缓冲器是很有必要的，因为 PIC18 不能提供足够的电流。如图 C-14 所示。

738

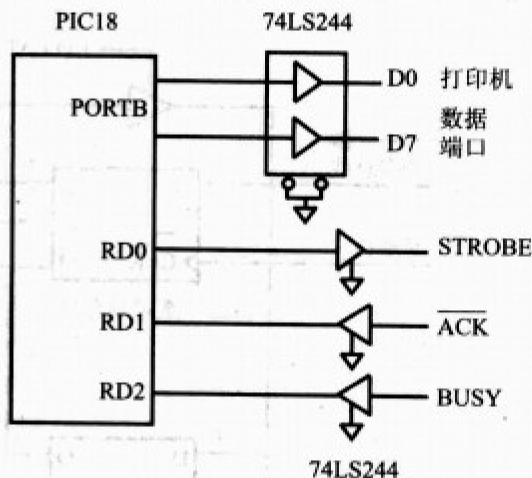


图 C-14　PIC18 连接到打印机的信号

C.3　系统设计问题

除了扇出能力之外，关于系统设计的其他问题包括：能耗、接地抖动、V_{cc} 抖动、串扰和传输线。在本节中将对这些问题做一个概述。

C.3.1　能耗的考虑

系统的能耗是系统设计者考虑的主要问题，特别是使用电池供电的手提电脑和掌上设备。能耗是关于频率和电压的函数，如下所示：

$$Q = CV$$

$$\frac{Q}{T} = \frac{CV}{T}$$

因为　$F = \frac{1}{T}$　并且　$I = \frac{Q}{T}$

$$I = CVF$$

所以　$P = VI = CV^2F$

在上面的方程中，要注意频率和V_{cc}电压的影响。当能耗随频率线性地增长时，电源的影响更加地明显（二次方曲线）。请看例C-2。

例C-2 比较两个基于微控制器的系统的能耗。一个使用5 V的V_{cc}，而另一个则使用3 V的V_{cc}。

解：因为$P=VI$，代入$I=V/R$，得到$P=V^2/R$。假设$R=1$，于是有$P=5^2=25$ W和$P=3^2=9$ W。两者的能耗相差16 W，也就是对于使用3 V电压源供电的系统来说，节省了64%的功率。（16/25×100=64%）。

C.3.2 动态电流和静态电流

流过IC的电流类型主要有两种：动态电路和静态电路。动态电流是$I=CVF$，它是元件工作频率的函数。也就是，当频率增加时，动态电流和能耗将会上升。静态电流（即DC）是元件非活动时（没有选用）的电流损耗。动态电流损耗比静态电流损耗要大得多。为了减少能耗，很多微控制器（包括PIC18）都设有节能模式。对于PIC18，节能模式又叫休眠模式。下面介绍休眠模式。

739

休眠模式

在休眠模式下，片内振荡器停止工作，切断CPU和外围功能部件（如串行端口、中断以及定时器）的时钟信号。注意，在休眠模式将能耗降到绝对最低值时，RAM和SFR寄存器的内容依然保持不变。

C.3.3 接地抖动

高频系统设计者要面对的一个主要问题是接地抖动。在定义接地抖动之前，先来讨论IC引脚的感应系数。毫无疑问，在IC的每个引脚上都有一定的电容、电阻和电感。这些要素的大小取决于很多因素，如长度、面积等。

引脚的电感系数通常称为自感系数，这同下面将要介绍的互感系数相区别。在电容、电阻和电感三个要素中，影响高频系统设计最严重的是电感，因为它会导致接地抖动。当很多输出引脚同时从高电平变成低电平、进而导致大量电流流入接地引脚时，就会出现接地抖动，如图C-15a所示。电压与接地电感的关系如下所示：

$$V=L\frac{di}{dt}$$

当增加系统频率时，动态电流的速率di/dt也会增加，这将导致接地引脚的自感电压$L(di/dt)$也增加。因为低电平状态（接地）的噪声裕度较小，由自感引起的任何额外电压都可能会导致错误的信号。为了降低接地抖动的影响，应尽可能地采取下面的步骤。

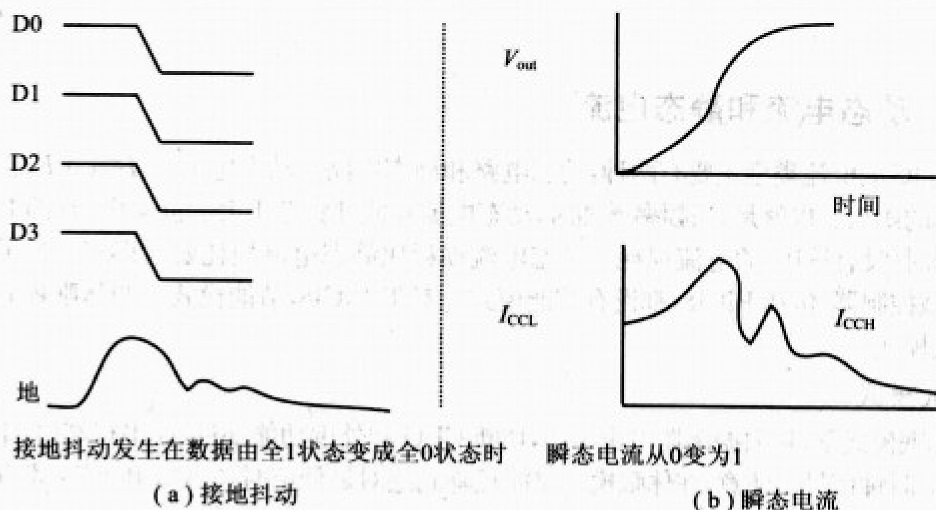
(1) V_{cc}和接地引脚要安排在IC芯片的中间位置而不是相反的两端（14脚的TTL逻辑IC使用引脚14和引脚7作为接地和V_{cc}）。这在高性能的逻辑门电路中可以看到，如德州仪器生产的高级逻辑电路AC11000和ACT11000。例如，ACT11013是一个14引脚的DIP芯片，引脚4和11用作接地和V_{cc}，取替了传统TTL系列用引脚7和引脚14的做法。也可以使用SOIC封装代替DIP封装。

(2) 另一个解决方法是使用尽量多的接地引脚和V_{cc}引脚来减少导线长度。这就是为什

么所有的高性能微处理器和逻辑器件系列都用很多的 V_{cc} 引脚和接地引脚，而不是传统地采用单一的 V_{cc} 和接地引脚。例如，在因特尔的奔腾处理器上就有超过 50 个接地引脚和 50 个 V_{cc} 引脚。

740

上面关于接地抖动的讨论同样适用于 V_{cc}。当大量的输出引脚由低电平变成高电平时，就会出现所谓的 V_{cc} 抖动。然而，V_{cc} 抖动的影响没有接地抖动那么严重，因为高("1")电平状态相比低("0")电平有更大的噪声裕度。



(a) 接地抖动 (b) 瞬态电流

图 C-15

C.3.4 用去耦电容滤除瞬态电流

在 TTL 器件里，输出端从低电平变成高电平时就会导致*瞬态电流*的产生。对于输出为低电平的推拉式输出，Q4 导通且饱和导通，而 Q3 截止。当输出从低电平变成高电平时，Q3 导通而 Q4 截止。也就是说有这样一个时刻存在，即所有晶体管都导通而且从 V_{cc} 吸收电流。电流量的大小由两个晶体管的 R_{ON} 决定，也就是由晶体管的内部参数决定。这种影响对输出形成一个很大的脉冲电流，如图 C-15b 所示。为了滤除瞬态电流，可以在每个 TTL 型 IC 的 V_{cc} 引脚和接地引脚之间加接一个 0.01 μF 或者 0.1 μF 的陶瓷电容。不过，所接电容的导线要尽可能地短，因为长的导线会导致大的自感系数，从而会在 V_{cc} 线路上产生尖峰脉冲。这个脉冲就叫作 V_{cc} 抖动。IC 芯片上的陶瓷电容被称为*去耦电容*。下面还将介绍一种大容量的去耦电容。

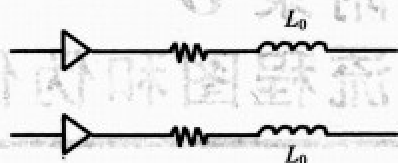
C.3.5 大容量去耦电容

如果很多 IC 芯片在同一时间改变状态，电路板上 V_{cc} 提供的总电流将会很大，有可能导致电路板上 V_{cc} 的波动。为了消除这种情况，可以在 V_{cc} 和接地之间放置一个大容量的去耦钽电容。钽电容的大小和位置由电路板上的 IC 数目和总的电流决定，不过，对于 16 芯片的电路系统，通常需要在每个芯片的 V_{cc} 和接地之间放置一个 22～47 μF 的电容。

741

C.3.6 串扰

串扰由互感造成，如图C-16所示。前面已讨论过电感中的自感。互感是由两条平行的导线相互作用而引起的。互感系数是l（两个平行导线的长度）、d（电感间的距离）和中间介质的函数。拉开两条平行或者相邻的导线（在PCB中称为布线）之间的距离，可以降低串扰的影响。在很多场合（如打印机和磁盘驱动电缆）中，每条信号线都有对应的接地线。在信号线之间布置地线可以减低串扰的作用。该方法也用在一些ACT逻辑器件中，因为它们的V_{cc}引脚和GND引脚隔得很近。串扰又被称为EMI（电磁干扰）。这是相对于ESI（静电干扰）来说的，后者是由相连的两个导体的电容耦合引起的。

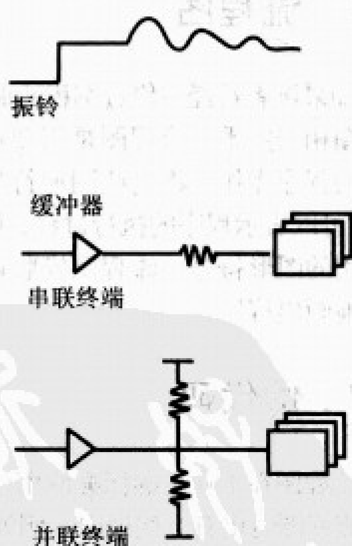


图C-16 串扰（EMI）

C.3.7 传输线振铃

在数字电路里使用的方波实际上就是由单个基本脉冲和不同幅值的谐波组成的。当该信号在线路上传输时，不是所有的谐波信号都对线路上的电容、电阻、电感有相同的响应。这样就会产生振铃现象，它由线路驱动器的厚度和长度以及其他因素决定。为了降低振铃的影响，往往会在线路驱动器的末端连接一个电阻，如图C-17所示。线路驱动终端的连接主要有三种类型：并联、串联和戴维宁。

在串联终端中，在线路的末端串接30～50Ω的电阻。在并联和戴维宁终端线路中，线路的阻抗需要同负载阻抗相匹配。这就需要详细地分析信号走线和负载阻抗，显然这已超出本书的范围。在高频系统中，布置在印制电路板（PCB）上的走线跟传输线一样，也会导致振铃现象。振铃的严重程度同所使用的逻辑器件速度有关。表C-7给出了参考的走线长度，超过这个参考值的走线要当作传输线对待。



图C-17 降低传输线振铃的方法

742

表C-7 当作传输线的走线的线路长度

逻辑器件	线路长度
LS	25
S，AS	11
F，ACT	8
AS，ECL	6
FCT，FCTA	5

本转载经由IDT公司许可，IDT版权保护，1991年。

743

附录 D 流程图和伪代码

概述

本附录将介绍流程图和伪代码的写法。

D.1 流程图

如果读者曾经主修过编程的课程，那么就可能对流程图相当熟悉。流程图是用图形符号来表示不同类型的程序操作。将这些图形符号连接成一个流程图，就可以表示程序的执行过程。图 D-1 给出了一些最常用的图形符号。流程图模版可以让读者快速而整洁地画出符号。

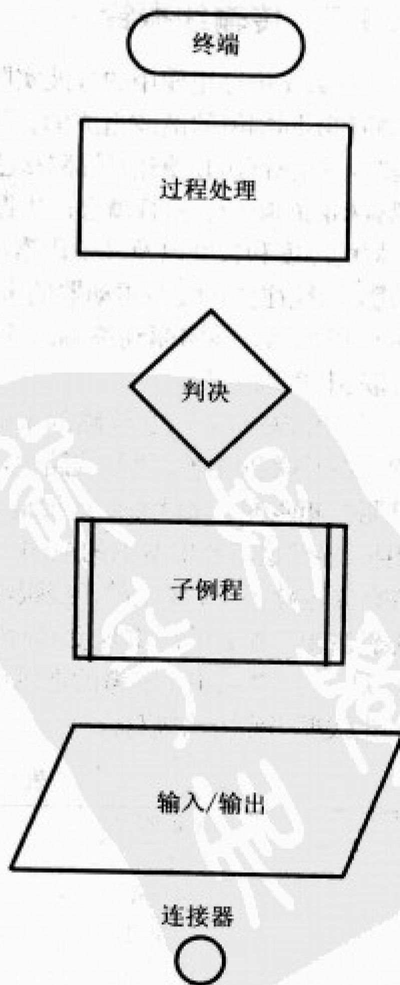


图 D-1 常用的流程图符号

D.2 伪代码

流程图在工业上的标准使用已有数十年。然而，使用流程图时仍有一些局限。例如，读者不可能在一个小方框里写入很多东西，而且很难在不打开细节的情况下直观地了解程序的执行内容。一种改进的方法就是在流程图中使用伪代码，也就是简要地描述代码的流程。图 D-2～图 D-6 给出了常用控制结构的流程图和伪代码。

结构化程序设计使用 3 种程序控制结构：顺序、控制和循环。顺序结构就是一条接一条地、简单地执行程序。图 D-2 绘出了用伪代码和流程图表示的顺序结构。

```
Statement 1
Statement 2
```

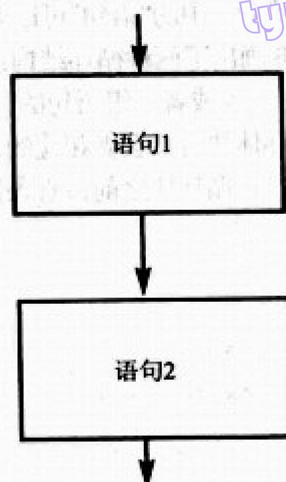


图 D-2 顺序结构的伪代码与流程图

图 D-3 和图 D-4 用伪代码和流程图绘出了两个程序控制结构：IF-THEN-ELSE 746 和IF-THEN。

```
IF (condition) THEN
     Statement 1
ELSE
     Statement 2
```

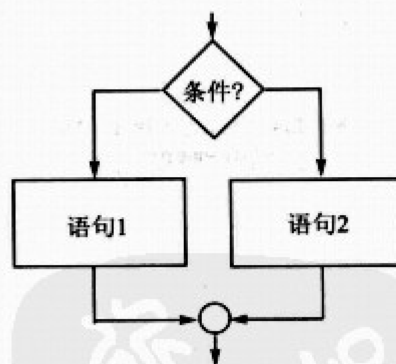


图 D-3 IF THEN ELSE 伪代码和流程图

```
IF (condition) THEN
     Statement
```

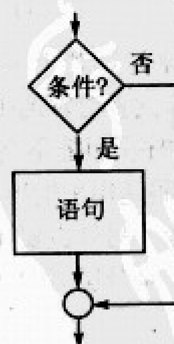


图 D-4 IF THEN 伪代码和流程图

注意,图 D-2~图 D-6 中的"语句"可以表示一条或者一组语句。

图 D-5 和图 D-6 画出了两种循环结构:REPEAT UNTIL 和 WHILE DO。这两种结构的功能都是重复执行一条或者一组语句的。它们的区别是,REPEAT UNTIL 结构至少执行语句一次,并且在循环体执行后会做相应的条件检查;而 WHILE DO 可能根本不会执行循环体内的语句,因为它是在循环体之前检查条件的。

747

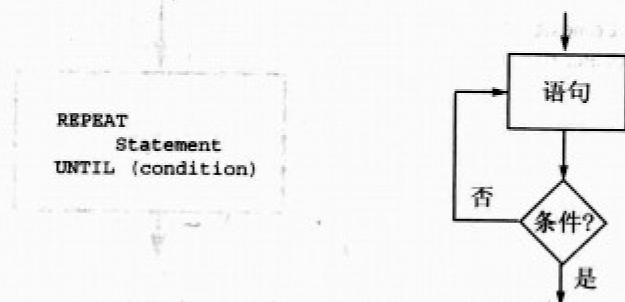


图 D-5 REPEAT UNTIL 伪代码和流程图

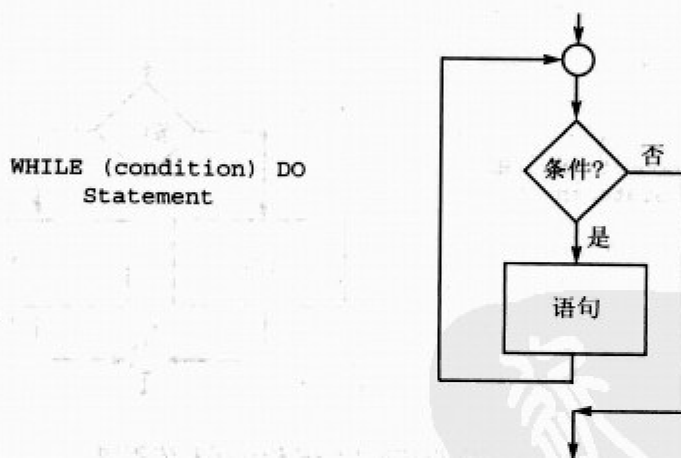


图 D-6 WHILE DO 伪代码和流程图

程序 D-1 是用来计算若干字节数的算术和。比较程序的流程图和伪代码(如图D-7所示)。在这个例子中,读者会发现比平常更多的程序细节。例如,它给出了计数器初始化和减1的步骤。其他的程序员可能不会把这些步骤写入流程图或者伪代码中。要记住很重要的一点是,流程图和伪代码是用来给出程序的流程和程序所要实现的功能,而不是实现程序目标的特定汇编语言指令。另外也要注意,伪代码给出的信息比流程图给出的相同信息要更复杂一些。有时候伪代码是分层写的,所以外层表示程序流程,内层体现在完成工作时的更多程序细节。

748

```
Count = 5
Address = 40H
Repeat
      Add next byte
      Increment address
      Decrement counter
Until Count = 0

Store Sum
```

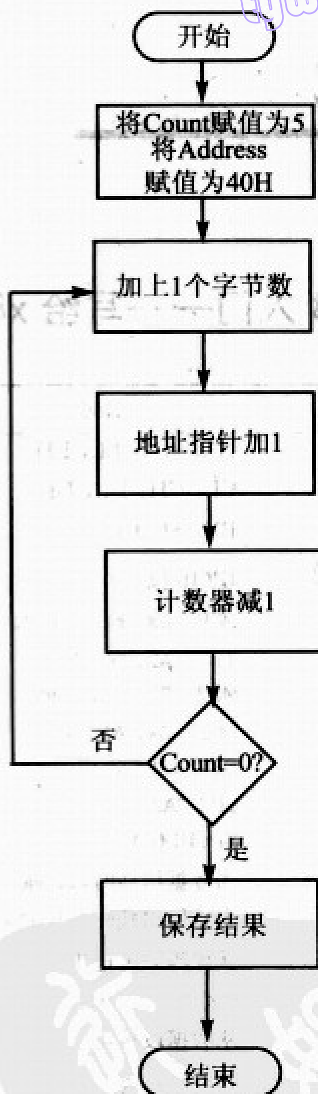


图 D-7 程序 D-1 的伪代码和流程图

程序 D-1

```
COUNTVAL    EQU  5              ;COUNT = 5
COUNTREG    SET  0x20           ;set aside location 20H for counter
SUM         SET  0x30           ;set aside location 30H for sum
       MOVLW  COUNTVAL          ;WREG = 5
       MOVWF  COUNTREG          ;load the counter
       LFSR   0,0x40            ;load pointer. FSR0 = 40H, RAM address
       CLRF   WREG              ;clear WREG
B5     ADDWF  POSTINC0,.W       ;add RAM to WREG and increment FSR0
       DECF   COUNTREG,F        ;decrement counter
       BNZ    B5                ;loop until counter = zero
       MOVWF  SUM               ;store WREG in SUM
```

附录 E

E.1 PIC18 入门——写给 x86 程序员

	X86	PIC18
8 位寄存器	AL, AH, BL, BH, CL, CH, DL, DH	WREG 寄存器和多达 256 个 RAM 地址的访问寄存区
16 位(数据指针)	BX, SI, DI	TBLPTR
程序计数器	IP(16 位)	PC(21 位)
输入	MOV DX, port addr IN AL, DX	MOVFW PORTx; (x= A,B,…G)
输出	MOV DX, port addr OUT DX, AL	MOVWF PORTx; (x= A,B,…G)
循环	DEC CL JNZ TARGET	DECF MyReg, F BNZ TARGET
栈指针	SP(16 位) 当数据压栈时,SP 减 1 当数据出栈时,SP 加 1	SP(21 位) 压栈时 SP 加 1(专门用于保存 PC) 出栈时 SP 减 1(专门用于恢复 PC)
数据传送	从代码段传送出: MOV AL, CS:[SI]	TBLRD
	从数据段传送出: MOV AL, [SI]	MOVFW FSRx
	从 RAM 传送到: MOV AL, [SI] (仅使用 SI、DI 或者 BX)	MOVFW FSRx
	传送到 RAM: MOV [SI], AL	MOVWF FSRx

E.2 PIC18 入门——写给 8051 程序员

	8051	PIC18
8 位寄存器	A,B,R0,R1,…,R7	WREG 和多达 256 个 RAM 地址的访问寄存区
16 位(数据指针)	DPTR	TBLPTR

（续）

	8051	PIC18
程序计数器	PC(16 位)	PC(21 位)
输入	MOV A, Pn; (N= 0~3)	MOVFW PORTx; (x= A,B,…G)
输出	MOV Pn, A; (N= 0~3)	MOVWF PORTx; (x= A,B,…G)
循环	DJNZ R3, TARGET (使用 R0~R7)	DECF MyReg, F BNZ TARGET
栈指针	SP(8 位) 当数据压栈时,SP 减 1 当数据出栈时,SP 加 1	SP(21 位) 压栈时 SP 加 1(专门用于保存 PC) 出栈时 SP 减 1(专门用于恢复 PC)
数据传送	从代码段传送出: MOVC A, @A+ PC 从数据段传送出: MOVX A, @DPTR 从 RAM 传送出: MOV A, @R0 (仅使用 R0 和 R1) 传送到 RAM: MOV @R0, A	 TBLRD MOVFW FSRx MOVFW FSRx MOVWF FSRx

附录 F

ASCII 码

Ctrl	Dec	Hex	Ch	Code	Dec	Hex	Ch	Dec	Hex	Ch	Dec	Hex	Ch
^@	0	00		NUL	32	20		64	40	@	96	60	`
^A	1	01	☺	SOH	33	21	!	65	41	A	97	61	a
^B	2	02	☻	STX	34	22	"	66	42	B	98	62	b
^C	3	03	♥	ETX	35	23	#	67	43	C	99	63	c
^D	4	04	♦	EOT	36	24	$	68	44	D	100	64	d
^E	5	05	♣	ENQ	37	25	%	69	45	E	101	65	e
^F	6	06	♠	ACK	38	26	&	70	46	F	102	66	f
^G	7	07	•	BEL	39	27	'	71	47	G	103	67	g
^H	8	08	◘	BS	40	28	(	72	48	H	104	68	h
^I	9	09	○	HT	41	29	)	73	49	I	105	69	i
^J	10	0A	◙	LF	42	2A	*	74	4A	J	106	6A	j
^K	11	0B	♂	VT	43	2B	+	75	4B	K	107	6B	k
^L	12	0C	♀	FF	44	2C	,	76	4C	L	108	6C	l
^M	13	0D	♪	CR	45	2D	-	77	4D	M	109	6D	m
^N	14	0E	♫	SO	46	2E	.	78	4E	N	110	6E	n
^O	15	0F	☼	SI	47	2F	/	79	4F	O	111	6F	o
^P	16	10	►	DLE	48	30	0	80	50	P	112	70	p
^Q	17	11	◄	DC1	49	31	1	81	51	Q	113	71	q
^R	18	12	↕	DC2	50	32	2	82	52	R	114	72	r
^S	19	13	‼	DC3	51	33	3	83	53	S	115	73	s
^T	20	14	¶	DC4	52	34	4	84	54	T	116	74	t
^U	21	15	§	NAK	53	35	5	85	55	U	117	75	u
^V	22	16	▬	SYN	54	36	6	86	56	V	118	76	v
^W	23	17	↨	ETB	55	37	7	87	57	W	119	77	w
^X	24	18	↑	CAN	56	38	8	88	58	X	120	78	x
^Y	25	19	↓	EM	57	39	9	89	59	Y	121	79	y
^Z	26	1A	→	SUB	58	3A	:	90	5A	Z	122	7A	z
^[	27	1B	←	ESC	59	3B	;	91	5B	[	123	7B	{
^\	28	1C	∟	FS	60	3C	<	92	5C	\	124	7C	|
^]	29	1D	↔	GS	61	3D	=	93	5D	]	125	7D	}
^^	30	1E	▲	RS	62	3E	>	94	5E	^	126	7E	~
^_	31	1F	▼	US	63	3F	?	95	5F	_	127	7F	⌂

Dec	Hex	Ch	Dec	Hex	Ch	Dec	Hex	Ch	Dec	Hex	Ch
128	80	Ç	160	A0	á	192	C0	└	224	E0	α
129	81	ü	161	A1	í	193	C1	┴	225	E1	ß
130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
131	83	â	163	A3	ú	195	C3	├	227	E3	π
132	84	ä	164	A4	ñ	196	C4	─	228	E4	Σ
133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
134	86	å	166	A6	ª	198	C6	╞	230	E6	µ
135	87	ç	167	A7	º	199	C7	╟	231	E7	τ
136	88	ê	168	A8	¿	200	C8	╚	232	E8	Φ
137	89	ë	169	A9	⌐	201	C9	╔	233	E9	Θ
138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
141	8D	ì	173	AD	¡	205	CD	═	237	ED	φ
142	8E	Ä	174	AE	«	206	CE	╬	238	EE	ε
143	8F	Å	175	AF	»	207	CF	╧	239	EF	∩
144	90	É	176	B0	░	208	D0	╨	240	F0	≡
145	91	æ	177	B1	▒	209	D1	╤	241	F1	±
146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
147	93	ô	179	B3	│	211	D3	╙	243	F3	≤
148	94	ö	180	B4	┤	212	D4	╘	244	F4	⌠
149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	°
153	99	Ö	185	B9	╣	217	D9	┘	249	F9	∙
154	9A	Ü	186	BA	║	218	DA	┌	250	FA	·
155	9B	¢	187	BB	╗	219	DB	█	251	FB	√
156	9C	£	188	BC	╝	220	DC	▄	252	FC	ⁿ
157	9D	¥	189	BD	╜	221	DD	▌	253	FD	²
158	9E	₧	190	BE	╛	222	DE	▐	254	FE	■
159	9F	ƒ	191	BF	┐	223	DF	▀	255	FF	

附录 G 编译器、开发资源和供应商

本附录将介绍许多关于 PIC18 编译器和调试器的开发资源。另外，还列出了一些芯片和其他硬件的供应商。由于这些是知名公司的成熟产品，本书作者和出版商将不会为它们引起的任何问题承担责任。作者不鼓励也不反对读者购买书中所提到的产品，在评估产品时读者应有自己的判断，这些列表仅供读者参考。另外，还需要说明的是，表中所列的产品是不完整的。

G.1 PIC18 编译器

PIC18 编译器由 Microchip 公司和其他的许多公司提供。有些公司专门提供了产品的共享软件，读者可以从它们的网站上免费下载。不过，这些共享软件的代码容量被限制为几个 KB。图 G-1 列出了编译器的部分供应商。

Microchip 公司
www. microchip. com

客户计算机服务公司
www. ccsinfo. com

图 G-1 汇编器和编译器的供应商

G.2 PIC18 调试器

目前有很多公司生产和销售 PIC18 调试器。图 G-2 列出了提供 PIC 调试器的部分厂商。

Microchip 公司
www. microchip. com

www. MicroDigitalEd. com

客户计算机服务公司
www. ccsinfo. com

RSR 电子
www. elexp. com

图 G-2 调试器供应商

G.3 电子器件供应商

图 G-3 列出了许多电子器件的供应商。

RSR 电子
Electronix Express
365 Blair Road
Avenel, NJ 07001
传真:(732)381 - 1572
邮购:1 - 800 - 972 - 2225
新泽西:(732)381 - 8020
www.elexp.com

Altex 电子
11342 IH - 35 North
San Antonio, TX 78233
传真:(210) 637 - 3264
邮购:1 - 800 - 531 - 5369
www.altex.com

Digi - Key
1 - 800 - 344 - 4539 (1 - 800 - DIGI - KEY)
传真:(218) 681 - 3380
www.digikey.com

Radio Shack
www.radioshack.com

JDR Microdevices
1850 South 10th St.
San Jose, CA 95112 - 4108
销售电话:1 - 800 - 538 - 5000
(408)494 - 1400
传真:1 - 800 - 538 - 5005
传真:(408) 494 - 1420
www.jdr.com

Mouser Electronics
958 N. Main St.
Mansfield, TX 76063
1 - 800 - 346 - 6873
www.mouser.com

Jameco Electronic
1355 Shoreway Road
Belmont, CA 94002 - 4100
1 - 800 - 831 - 4242
(415)592 - 8097
传真:1 - 800 - 237 - 6948
传真:(415)592 - 2503
www.jameco.com

B. G. Micro
P. O. Box 280298
Dallas, TX 75228
1 - 800 - 276 - 2206 (仅用于订货)
(972)271 - 5546
传真:(972) 271 - 2462
这是一家供应 LCD、IC、按键等的优秀供应商。
www.bgmicro.com

Tanner Electronics
1100 Valwood Parkway, Suite #100
Carrollton, TX 75006
(972)242 - 8702
www.tannerelectronics.com

图 G-3 电子器件的供应商

附录 H
数据表——PIC18F2480/2580/4480/4580

H.0 指令集简介

PIC18F2480/2580/4480/4580 设备的指令系统在 PIC18 的 75 条标准的核心指令基础上，扩展了 8 条新的指令，用于优化递归的程序代码或者利用软件栈的程序。扩展的指令集将在本节稍后讨论。

H.1 标准指令集

标准的 PIC18 指令集，在保持与以前 PICmicro ® 指令集兼容的同时，还加入了很多改进的指令。大部分指令只占用程序存储器的一个字(16 位)空间，但是还有 4 条指令是占用两个程序存储器地址的。

每一个单字指令都是 16 位的，并被细分为操作码和一个或多个操作数。操作码用来指明指令的类型，而操作数用来进一步指明指令的操作。

这些指令集是高度正交的，可以分成 4 个基本的类型：

□ **面向字节的操作**

□ **面向位的操作**

□ **立即数操作**

□ **控制操作**

表 H-2 列出了 PIC18 指令集的**面向字节**的操作类指令、**面向位**的操作类指令、**立即数**操作类指令和**控制**操作类指令。表 H-1 还给出了操作码段说明。

大多数**面向字节**的操作类指令由 3 个操作数组成：

□ 文件寄存器(由 f 的值来指定)

□ 目的寄存器(由 d 的值来指定)

□ 可寻址的存储器(由 a 的值来指定)

f 表示指定的文件寄存器，文件寄存器说明指令使用哪个寄存器。d 表示选中的目的寄存器，目的寄存器指定了操作的结果所存放的位置。如果 d 为 0，结果保存于 WREG 寄存器。

如果 d 为 1,结果保存于指定的文件寄存器。

所有面向位的操作类指令都由 3 个操作数组成:

□文件寄存器(由 f 的值来指定)

□文件寄存器的位(由 b 的值来指定)

□可寻址的存储器(由 a 的值来指定)

b 表示操作所涉及的寄存器的哪一位,而 f 表示该位所在的文件寄存器地址。

立即数操作类指令由如下的几个操作数构成:

□要放入文件寄存器的立即数(由 k 的值来指定)

□装入立即数的 FSR 寄存器(由 f 的值来指定)

□没有操作数(由—来指定)

控制操作类指令由如下的几个操作数构成:

□程序存储器地址(由 n 的值来指定)

□CALL 或者 RETURN 指令的模式(由 s 的值来指定)

□表读取或者表写入指令的模式(由 m 的值来指定)

□没有操作数(由"—"来指定)

除了 4 个双字指令之外,所有指令都是单字的。这些指令被设计成双字是为了满足 32 位的要求。在第二字中,最高的 4 位都是 1。如果第二字是当作一个指令来执行,那么它相当于执行了 NOP 指令。

所有单字指令占用 1 个单指令周期,除非出现条件测试为"真"或者程序计数器因其他指令结果改变了 PC 的值。如果是那样的话,在执行时会占用两个指令周期,而在多出的第二个指令周期中执行的是一条 NOP 指令。

双字指令的执行需要两个指令周期。、

每个指令周期由 4 个晶振周期组成。因此,对于 4 MHz 的晶振频率,通常指令执行时间为 1 μs。如果条件测试为"真",或者程序计数器 PC 的值被另一条指令改变,那么指令执行时间就为 2 μs。两字长的分支指令(如果条件为"真")将会占用 3 μs。

图 H-1 列出了指令的常用格式。所有例子都约定使用 nnh 来表示一个十六进制数。

表 H-2 是指令集的总结,列出了 Microchip MPASM™编译器可识别的标准指令。

在 H.1.1 节中还给出了每一条指令的描述。

756
~
757

表 H-1 操作码段说明

段	描 述
a	RAM 的访问标识位 a=0,可以访问当前 RAM(忽略 BSR 寄存器) a=1,由 BSR 的值来指定 RAM 存储区
bbb	8 位文件寄存器(0~7)内的位地址
BSR	存储区选择寄存器,用于选择当前的存储区
C, DC, Z, OV, N	ALU 的状态位:进位,半进位,零,溢出,负数
d	表示目的寄存器选择位 d=0,结果保存在 W 寄存器中 d=1,结果保存在文件寄存器 f 中

（续）

段	描　述
dest	表示目的地址：要么是W寄存器，要么是指导的文件寄存器f的地址
f	8位文件寄存器地址(00h~FFh)，或者2位FSR指示器(0h~3h)
f_s	12位文件寄存器地址(000h~FFFh)，表示源地址
f_d	12位文件寄存器地址(00h~FFh)，表示目的地址
GIE	全局中断使能位
k	立即数、常量或者标号(可以是8位、12位或者16位的值)
label	标号名
mm	TBLPTR寄存器模式的表读取和表写入指令 只有在执行读取和写入指令时才使用
*	不更改寄存器(如TBLPTR表读取/表写入)
*+	后增量寄存器(如TBLPTR表读取/表写入)
*–	后减量寄存器(如TBLPTR表读取/表写入)
+*	前增量寄存器(如TBLPTR表读取/表写入)
n	相对分支转移指令的相对地址(二进制补码数)，或者用于调用/分支和返回指令的直接地址
PC	程序计数器
PCL	程序计数器的低字节
PCH	程序计数器的高字节
PCLATH	程序计数器的高字节锁存器
PCLATU	程序计数器的更高字节锁存器
$\overline{PD}$	掉电标志位
PRODH	乘法运算的积的高字节
PRODL	乘法运算的积的低字节
s	快速调用/返回模式选择位 s=0：不能读取或者写入影子寄存器 s=1：某些寄存器读取或者写入影子寄存器(快速模式)
TBLPTR	21位表指针(指向程序存储器的地址)
TABLAT	8位表锁存器
$\overline{TO}$	"定时时间到"位
TOS	栈顶
u	未使用或者未变化
WDT	看门狗定时器
WREG	工作寄存器(累加器)
x	不考虑(0或者1)。这种情况用于Microchip软件工具的兼容，在编译器中它会产生一个为零的代码
Z_s	7位偏移地址，用于寄存器文件的间接寻址(源操作数)
Z_d	7位偏移地址，用于寄存器文件的间接寻址(目的操作数)
{ }	表示可选项
[text]	表示索引地址
(text)	表示是text的内容
[expr]<n>	指针expr所指向的寄存器的二进制位n
→	定向选择赋值
< >	表示寄存器位的范围

（续）

段	描　述
∈	表示包含于
Italics	用户定义项(字体为 Courier)

面向字节的文件寄存器操作　　　　指令举例

15　10　9　8　7　0

操作码	d	a	f(文件寄存器#)

ADDWF MYREG, W, B

d=0:结果保存在 WREG 寄存器中

d=1:结果保存在文件寄存器中

a=0:仅限于访问存储区

a=1:由 BSR 来选择存储区

f=8 位文件寄存器地址

从字节到字节的传送操作(2 字)

15　12　11　0

操作码	f(源文件寄存器#)

MOVFF MYREG1, MYREG2

15　12　11　0

1111	f(目的文件寄存器#)

f=12 位文件寄存器地址

面向位的文件寄存器操作

15　12　11　9　8　7　0

操作码	d(位#)	a	f(文件寄存器#)

BSF MYREG,bit,B

b=文件寄存器 f 的 3 个二进制位

a=0:仅限于访问存储区

a=1:由 BSR 来选择存储区

f=8 位文件寄存器地址

立即数的操作

15　8　7　0

操作码	k(文件寄存器#)

MOVLW 7Fh

k=8 位文件寄存器地址

控制操作

CALL,GOTO 和分支转移操作

15　8　7　0

操作码	n<7:0>(立即数)

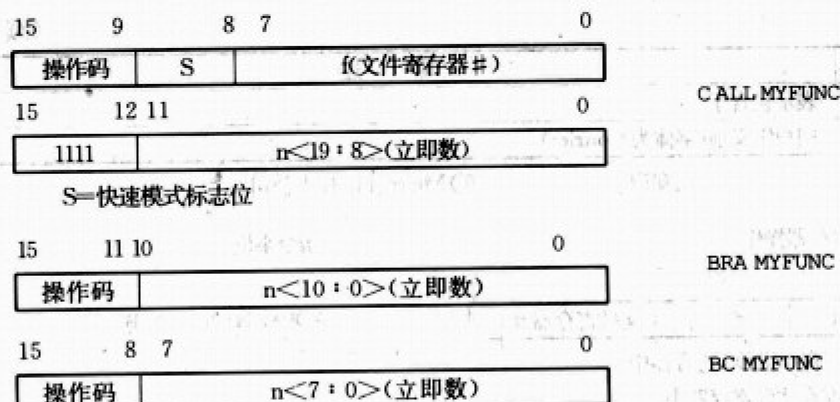
GOTO Label

15　12　11　0

操作码	n<19:8>(立即数)

n=20 位立即数

图 H-2　通用的指令格式



CALL MYFUNC

BRA MYFUNC

BC MYFUNC

图 H-2 (续)

表 H-2 PIC18FXXXX 指令集

助记符,操作数	描 述	周期	16 位指令字 MSB LSB	影响的标志位	备注
面向字节的操作类指令					
ADDWF f, d, a	将 WREG 与 f 相加	1	0010 01da ffff ffff	C, DC, Z, OV, N	1, 2
ADDWFC f, d, a	将 WREG 与 f 相加,带进位,结果存入 f 中	1	0010 00da ffff ffff	C, DC, Z, OV, N	1,2
ANDWF f, d, a	WREG 逻辑与 f	1	0001 01da ffff ffff	Z, N	1,2
CLRF f, a	清零 f	1	0110 101a ffff ffff	Z	2
COMF f, d, a	求 f 的补码	1	0001 11da ffff ffff	Z, N	1,2
CPFSEQ f, a	将 f 与 WREG 比较,若等于,跳过	1(2 或 3)	0110 001a ffff ffff	无	4
CPFSGT f, a	将 f 与 WREG 比较,若大于,跳过	1(2 或 3)	0110 010a ffff ffff	无	4
CPFSLT f, a	将 f 与 WREG 比较,若小于,跳过	1(2 或 3)	0110 000a ffff ffff	无	1,2
DECF f, d, a	f 减 1 操作	1	0000 01da ffff ffff	C, DC, Z, OV, N	1,2,3,4
DECFSZ f, d, a	f 减 1 操作,若为零,跳过	1(2 或 3)	0010 11da ffff ffff	无	1,2,3,4
DCFSNZ f, d, a	f 减 1 操作,若不为零,跳过	1(2 或 3)	0100 11da ffff ffff	无	1,2
INCF f, d, a	f 加 1 操作	1	0010 10da ffff ffff	C, DC, Z, OV, N	1,2,3,4
INCFSZ f, d, a	F 加 1 操作,若为零,跳过	1(2 或 3)	0011 11da ffff ffff	无	4
INFSNZ f, d, a	F 加 1 操作,若为零,跳过	1(2 或 3)	0100 10da ffff ffff	无	1,2

（续）

助记符，操作数	描　述	周期	16 位指令字 MSB　LSB	影响的标志位	备注
面向字节的操作类指令					
IORWF f, d, a	WREG 逻辑或 f	1	0001 00da ffff ffff	Z, N	1,2
MOVF f, d, a	传送 f	1	0100 00da ffff ffff	Z, N	1
MOVFF f_s, f_d	将 fs 传送到第一个字 将 fd 传送到第二个字	2	1100 ffff ffff ffff 1111 ffff ffff ffff	无	
MOVWF f, a	将 WREG 传送到 f	1	0110 111a ffff ffff	无	
MULWF f, a	将 WREG 乘以 f	1	0000 001a ffff ffff	无	1,2
NEGF f, a	取反 f	1	0110 110a ffff ffff	C, DC, Z, OV, N	
RLCF f, d, a	寄存器 f 带进位循环左移 1 位	1	0011 01da ffff ffff	C, Z, N	1,2
RLNCF f, d, a	寄存器 f 不带进位循环左移 1 位	1	0100 01da ffff ffff	Z, N	
RRCF f, d, a	寄存器 f 带进位循环右移 1 位	1	0011 00da ffff ffff	C, Z, N	
RRNCF f, d, a	寄存器 f 不带进位循环右移 1 位	1	0100 00da ffff ffff	Z, N	
SETF f, a	置位 f	1	0110 100a ffff ffff	无	1,2
SUBFWB f,d,a	从 WREG 中减去 f，带借位	1	0101 01da ffff ffff	C, DC, Z, OV, N	
SUBWF f, d, a	从 f 中减去 WREG	1	0101 11da ffff ffff	C, DC, Z, OV, N	1,2
SUBWFB f,d,a	从 f 中减去 WREG，带借位	1	0101 10da ffff ffff	C, DC, Z, OV, N	
SWAPF f, d, a	f 的半字节交换操作	1	0011 10da ffff ffff	无	4
TSTFSZ f, a	测试 f，若为零，跳过	1（2 或 3）	0110 011a ffff ffff	无	1,2
XORWF f, d, a	WREG 逻辑异或 f	1	0001 10da ffff ffff	Z, N	
BCF f, b, a	清零 f 的位 b	1	1001 bbba ffff ffff	无	1,2
BSF f, b, a	置位 f 的位 b	1	1000 bbba ffff ffff	无	1,2
BTFSC f, b, a	测试 f 的位 b，若为零则跳过	1（2 或 3）	1011 bbba ffff ffff	无	3,4
BTFSS f,b,a	测试 f 的位 b，若为 1 则跳过	1（2 或 3）	1010 bbba ffff ffff	无	3,4
BTG f, b, a	对 f 寄存器的位取反	1	0111 bbba ffff ffff	无	1,2
控制操作类指令					
BC n	若 C=1，则运行分支子程序	1（2）	1110 0010 nnnn nnnn	无	

（续）

助记符,操作数	描　述	周期	16 位指令字 MSB			LSB	影响的标志位	备注
控制操作类指令								
BN n	若为负数,则运行分支子程序	1(2)	1110	0110	nnnn	nnnn	无	
BNC n	若 C=0,则运行分支子程序	1(2)	1110	0011	nnnn	nnnn	无	
BNN n	若不为负数,则运行分支子程序	1(2)	1110	0111	nnnn	nnnn	无	
BNOV n	若无溢出,则运行分支子程序	1(2)	1110	0101	nnnn	nnnn	无	
BNZ n	若无为 0,则运行分支子程序	1(2)	1110	0001	nnnn	nnnn	无	
BOV n	若有溢出,则运行分支子程序	1(2)	1110	0100	nnnn	nnnn	无	
BRA n	无条件执行分支子程序	2	1101	0nnn	nnnn	nnnn	无	
BZ n	若为 0,则运行分支子程序	1(2)	1110	0000	nnnn	nnnn	无	
CALL n, s	调用子程序,第一个字 第二个字	2	1110 1111	110s kkkk	kkkk kkkk	kkkk kkkk	无	
CLRWDT —	清零 WDT 定时器	1	0000	0000	0000	0100	$\overline{TO}$,$\overline{PD}$	
DAW —	WREG 的十进制调整	1	0000	0000	0000	0111	C	
GOTO n	转移到第一个字 第二个字	2	1110 1111	1111 kkkk	kkkk kkkk	kkkk kkkk	无	
NOP —	空操作	1	0000	0000	0000	0000	无	
NOP —	空操作	1	1111	xxxx	xxxx	xxxx	无	
POP —	弹出栈的顶端(TOS)	1	0000	0000	0000	0110	无	
PUSH —	压入栈的顶端(TOS)	1	0000	0000	0000	0101	无	
RCALL n	相对调用	2	1101	1nnn	nnnn	nnnn	无	
RESET	软件复位	1	0000	0000	1111	1111	所有标志位	
RETFIE s	从中断使能返回	2	0000	0000	0001	000s	GIE/GIEH, PEIE/GIEL	
RETLW k	立即数送入 WREG,返回	2	0000	1100	kkkk	kkkk	无	
RETURN s	从子程序返回	2	0000	0000	0001	001s	无	
SLEEP —	进入休眠状态	1	0000	0000	0000	0011	$\overline{TO}$,$\overline{PD}$	
立即数操作类指令								
ADDLW k	将立即数与 WREG 相加	1	0000	1111	kkkk	kkkk	C, DC, Z, OV, N	

(续)

助记符,操作数	描述	周期	16 位指令字 MSB			LSB	影响的标志位	备注
立即数操作类指令								
ANDLW k	将立即数与 WREG 做逻辑与操作	1	0000	1011	kkkk	kkkk	Z, N	
IORLW k	将立即数与 WREG 做逻辑或操作	1	0000	1001	kkkk	kkkk	Z, N	
LFSR f, k	传送 12 位立即数到 FSR,第一个字第二个字	2	1110 1111	1110 0000	00ff kkkk	kkkk kkkk	无	
MOVLB k	将立即数传送到 BSR <3:0>	1	0000	0001	0000	kkkk	无	
MOVLW k	将立即数传送到 WREG	1	0000	1110	kkkk	kkkk	无	
MULLW k	将立即数与 WREG 相乘	1	0000	1101	kkkk	kkkk	无	
RETLW k	立即数送 WREG,子程序返回	2	0000	1100	kkkk	kkkk	无	
SUBLW k	从 WREG 中减去立即数	1	0000	1000	kkkk	kkkk	C, DC, Z, OV, N	
XORLW k	将立即数与 WREG 做逻辑异或操作	1	0000	1010	kkkk	kkkk	Z, N	
数据存储器与程序存储器操作类指令								
TBLRD*	表读取	2	0000	0000	0000	1000	无	
TBLRD* +	带有后增量的表读取		0000	0000	0000	1001	无	
TBLRD* -	带有后减量的表读取		0000	0000	0000	1010	无	
TBLRD+ *	带有前增量的表读取		0000	0000	0000	1011	无	
TBLWT*	表写入	2	0000	0000	0000	1100	无	5
TBLWT* +	带有后增量的表写入		0000	0000	0000	1101	无	5
TBLWT* -	带有后减量的表写入		0000	0000	0000	1110	无	5
TBLWT+ *	带有前增量的表写入		0000	0000	0000	1111	无	5

注意:

(1)当端口寄存器作为它自身的一个功能修改时(如 MOVF PORTB, 1, 0),所使用的值将是当前在其引脚上的值。例如,对于用作输入的某个引脚,如果其数据锁存器为 1,且被外部器件的低电平所驱动,那么被写回的数据值将会是 0。

(2)如果前分频器分配给 TMR0 寄存器,那么任意一条对 TMR0 寄存器的操作指令都将使得前分频器清零。

(3)如果程序计数器(PC)被修改,或者条件测试为真,那么指令的执行需要 2 个周期。在第二个周期执行空操作。

(4)一些指令是 2 个字的指令。这些指令的第二个字将被当作一条 NOP 指令来执行,除非第一个字能检索到这 16 位中所包含的信息。这样,就保证了所有当前程序存储器地址都有一个有效的指令。

(5)如果开始向内部存储器进行表写入,那么这个表写入将一直持续到最后。

760 ~ 762

标准指令集

ADDLW 将立即数加到 WREG 中

格式： ADDLW k

操作数： 0≤k≤255

操作： (W)+k→W

受影响的标志位： N,OV,C,DC,Z

编码：

0000	1111	kkkk	kkkk

描述： 将 W 的内容和 8 位立即数相加，结果保存到 W。

字长： 1

周期： 1

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读立即数 k	处理数据	写 W 寄存器

例： ADDLW 15h

执行指令前，

W = 10h

执行指令后，

W = 25h

ADDWF 将 f 寄存器的内容与 WREG 寄存器的内容相加

格式： ADDWF f{,d{,a}}

操作数： 0≤f≤255

d∈[0,1]

a∈[0,1]

操作： (W)+(f)→(目的寄存器)

受影响的标志位： N,OV,C,DC,Z

编码：

0010	01da	ffff	ffff

描述： 将 W 寄存器的内容和 f 寄存器的内容相加。如果 d=0，结果放入 W。如果 d=1，结果放入 f 寄存器(默认)。

如果 a=0，选择当前访问存储区。如果 a=1，由 BSR 的值来选择 GPR 存储区(默认)。

如果 a=0 和扩展指令集可用，只要 f≤95(5Fh)，这条指令就会按立即数偏移量寻址模式执行。要获取更多的技术细节，请参阅 H.2.3 节。

字长： 1

周期： 1

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器 f	处理数据	写目的寄存器

例： ADDWF REG,0,0

执行指令前，

W = 17h

REG = 0C2h

执行指令后，

W = 0D9h

REG = 0C2h

注意：所有 PIC18 指令在指令助记符之前都可带有可选的标号。若使用标号，则指令格式将变成：{标号} 指令表达式。

ADDWFC 带进/借位的 WREG 和 f 相加

格式： ADDWFC f{,d{,a}}

操作数： 0≤f≤255
d∈[0,1]
a∈[0,1]

操作： (W)+(f)+(C)→目的寄存器

受影响的标志位： N,OV,C,DC,Z

编码：

0010	00da	ffff	ffff

描述： 将 W 寄存器的内容和进位标志位与 f 寄存器的内容相加。如果 d=0，结果放入 W。如果 d=1，结果放入 f 寄存器（默认）。
如果 a=0，当前的访问存储区被选择。如果 a=1，用 BSR 来选择 GPR 存储区（默认）。
如果 a=0 且扩展指令集可用，只要 f≤95(5Fh)，这条指令就会按立即数偏移量寻址模式执行。要获取更多的技术细节，请参阅 H.2.3 节。

字长： 1

周期： 1

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器 f	处理数据	写目的寄存器

例： ADDWFC REG, 0,1

执行指令前，
C = 1
REG = 02h
W = 4Dh

执行指令后，
C = 0
REG = 02h
W = 50h

ANDLW 立即数与 WREG 进行逻辑与操作

格式： ANDLW k

操作数： 0≤k≤255

操作： (W)按位与 k →W

受影响的标志位： N,Z

编码：

0000	1011	kkkk	kkkk

描述： 将 W 寄存器的内容和 8 位立即数做逻辑与运算，结果放入 W 寄存器

字长： 1

周期： 1

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读立即数 k	处理数据	写 W 寄存器

例： ANDLW 05Fh

执行指令前，
W = A3h

执行指令后，
W = 03h

ANDWF　　WREG 和 f 寄存器的内容做逻辑与操作

格式：　ANDWF f{,d{,a}}

操作数：　0≤f≤255

d∈[0,1]

a∈[0,1]

操作：　(W)按位与(f) → 目的寄存器

受影响的标志位：　N,Z

编码：

0001	01da	ffff	ffff

描述：　将 W 寄存器的内容和 f 寄存器的内容做逻辑与运算。如果 d=0，结果放入 W。如果 d=1，结果放入 f 寄存器（默认）。

如果 a=0，选择当前的访问存储区。如果 a=1，用 BSR 选择 GPR 存储区（默认）。

如果 a=0 且扩展指令集可用，只要 f≤95(5Fh)，这条指令就会按立即数偏移量寻址模式执行。要获取更多的技术细节，请参阅 H.2.3 节。

字长：　1

周期：　1

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器 f	处理数据	写目的寄存器

例：　ANDWF　REG, 0, 0

执行指令前，

W　=　17h

REG　=　C2h

执行指令后，

W　=　02h

REG　=　C2h

BC　　有进/借位就跳转

格式：　BC n

操作数：　-128≤k≤127

操作：　如果进/借位标志位为 1，(PC)+2+2n→ PC

受影响的标志位：　无

编码：

1110	0010	nnnn	nnnn

描述：　如果进/借位为 1，程序跳转。二进制补码 2n 被加到 PC 上。由于 PC 在取下一条指令时已经增加，所以新地址是(PC)+2+2n。这条指令需要两个指令周期。

字长：　1

周期：　1(2)

Q 周期活动：

如果跳转：

Q1	Q2	Q3	Q4
译码	读立即数 n	处理数据	写 PC
无操作	无操作	无操作	无操作

如果不跳转：

Q1	Q2	Q3	Q4
译码	读立即数 n	处理数据	无操作

例：　HERE　BC　5

执行指令前，

PC　=　地址(HERE)

执行指令后，

如果进位标志位　=　1

PC　=　地址(HERE+ 12)

如果进位标志位　=　0

PC　=　地址(HERE+ 2)

BCF　将 f 寄存器的位清零

格式：　BCF f,b{,a}

操作数：　0≤f≤255

0≤b≤7

a∈[0,1]

操作：　0 → f<b>

受影响的标志位：　无

编码：

1001	bbba	ffff	ffff

描述：　将 f 寄存器的 b 位清零。

如果 a=0，选择当前的访问存储区。如果 a=1，用 BSR 来选择 GPR 存储区（默认）。

如果 a=0 且扩展指令集可用，只要 f≤95(5Fh)，这条指令就会按立即数偏移量寻址模式执行。要获取更多的技术细节，请参阅 H. 2. 3 节。

字长：　1

周期：　1

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器 f	处理数据	写寄存器 f

例：　BCF　FLAG_REG, 7, 0

执行指令前，

FLAG_REG　=　C7h

执行指令后，

FLAG_REG　=　47h

BN　若为负数，则跳转

格式：　BN　n

操作数：　−128≤n≤127

操作：　如果是负数，

(PC)+2+2n→ PC

受影响的标志位：　无

编码：

1110	0110	nnnn	nnnn

描述：　如果负数标志位为 1，程序跳转。

二进制补码 2n 加到程序计数器上。因为 PC 在读取下一条指令时已经增加，所以新地址变成 PC+2+2n。这条指令需要 2 个指令周期。

字长：　1

周期：　1(2)

Q 周期活动：

如果跳转：

Q1	Q2	Q3	Q4
译码	读立即数 n	处理数据	写 PC
无操作	无操作	无操作	无操作

如果不跳转：

Q1	Q2	Q3	Q4
译码	读立即数 n	处理数据	无操作

例：　HERE　BN　Jump

执行指令前，

PC　=　地址(HERE)

执行指令后，如果是负数，

PC　=　地址(Jump)

如果不是负数，

PC　=　地址(HERE+ 2)

BNC 无进/借位则跳转

格式： BNC n

操作数： $-128 \leqslant n \leqslant 127$

操作： 如果进位标志位=0，
(PC)+2+2n→PC

受影响的标志位： 无

编码：

1110	0011	nnnn	nnnn

描述： 如果进/借位为0，程序跳转。二进制补码2n已经加到程序计数器上。因为PC在读取下一条指令时已经增加，新地址变成PC+2+2n。这条指令需要2个指令周期。

字长： 1

周期： 1(2)

Q周期活动：

如果跳转：

Q1	Q2	Q3	Q4
译码	读立即数n	处理数据	写PC
无操作	无操作	无操作	无操作

如果不跳转：

Q1	Q2	Q3	Q4
译码	读立即数n	处理数据	无操作

例： HERE BNC Jump

执行指令前，
PC = 地址(HERE)

执行指令后，
如果进位标志位 = 0
PC = 地址(Jump)
如果进位标志位 = 1
PC = 地址(HERE+2)

BNN 非负则跳转

格式： BNN n

操作数： $-128 \leqslant n \leqslant 127$

操作： 如果负数标志位=0，
(PC)+2+2n→PC

受影响的标志位： 无

编码：

1110	0111	nnnn	nnnn

描述： 如果负数标志位为0，程序跳转。二进制补码2n加到程序计数器上。因为PC在读取下一条指令时已经增加，新的地址变成PC+2+2n。这条指令需要2个指令周期。

字长： 1

周期： 1(2)

Q周期活动：

如果跳转：

Q1	Q2	Q3	Q4
译码	读立即数n	处理数据	写PC
无操作	无操作	无操作	无操作

如果不跳转：

Q1	Q2	Q3	Q4
译码	读立即数n	处理数据	无操作

例： HERE BNN Jump

执行指令前，
PC = 地址(HERE)

执行指令后，
如果负数标志位 = 0
PC = 地址(Jump)
如果负数标志位 = 1
PC = 地址(HERE+2)

BNOV 无溢出则跳转

格式： BNOV n

操作数： $-128 \leqslant n \leqslant 127$

操作： 如果溢出标志位=0，
(PC)+2+2n→PC

受影响的标志位： 无

编码：

1110	0101	nnnn	nnnn

描述： 如果溢出标志位为0，程序跳转。二进制补码2n加到程序计数器上。因为PC在读取下一条指令时已经增加，新的地址变成PC+2+2n。这条指令需要2个指令周期。

字长： 1

周期： 1(2)

Q周期活动：

如果跳转：

Q1	Q2	Q3	Q4
译码	读立即数 n	处理数据	写 PC
无操作	无操作	无操作	无操作

如果不跳转：

Q1	Q2	Q3	Q4
译码	读立即数 n	处理数据	无操作

例： HERE BNOV Jump

执行指令前，
PC = 地址(HERE)

执行指令后，
如果溢出标志位 = 0
PC = 地址(Jump)
如果溢出标志位 = 1
PC = 地址(HERE+ 2)

BNZ 非0则跳转

格式： BNZ n

操作数： $-128 \leqslant n \leqslant 127$

操作： 如果零标志位=0，
(PC)+2+2n→PC

受影响的标志位： 无

编码：

1110	0001	nnnn	nnnn

描述： 如果零标志位为1，程序跳转。二进制补码2n加到程序计数器上。因为PC在读取下一条指令时已经增加，新的地址变成PC+2+2n。这条指令需要2个指令周期。

字长： 1

周期： 1(2)

Q周期活动：

如果跳转：

Q1	Q2	Q3	Q4
译码	读立即数 n	处理数据	写 PC
无操作	无操作	无操作	无操作

如果不跳转：

Q1	Q2	Q3	Q4
译码	读立即数 n	处理数据	无操作

例： HERE BNZ Jump

执行指令前，
PC = 地址(HERE)

执行指令后，
如果零标志位 = 0
PC = 地址(Jump)
如果零标志位 = 1
PC = 地址(HERE+ 2)

BRA　　无条件跳转

格式：　BRA　n

操作数：　$-1024 \leqslant n \leqslant 1023$

操作：　(PC)+2+2n→PC

受影响的标志位：　无

编码：

1101	0nnn	nnnn	nnnn

描述：　将二进制补码 2n 加到程序计数器上。因为 PC 在读取下一条指时已经增加，新的地址变成 PC+2+2n。这条指令的执行需要 2 个指令周期。

字长：　1

周期：　2

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读立即数 n	处理数据	写 PC
无操作	无操作	无操作	无操作

例：　　HERE　　BRA　　Jump

执行指令前，

PC　=　地址(HERE)

执行指令后，

PC　=　地址(Jump)

BSF　　对 f 寄存器的位置 1

格式：　BSF f,b{,a}

操作数：　$0 \leqslant f \leqslant 255$

$0 \leqslant b \leqslant 7$

a∈[0,1]

操作：　0 → f<b>

受影响的标志位：　无

编码：

1000	bbba	ffff	ffff

描述：　对 f 寄存器的 b 位进行置 1 操作。

如果 a=0，选择当前的访问存储区。如果 a=1，用 BSR 来选择 GPR 存储区(默认)。

如果 a=0 且扩展指令集可用，只要 f≤95(5Fh)，这条指令就会按立即数偏移量寻址模式执行。要获取更多的技术细节，请参阅 H.2.3 节。

字长：　1

周期：　1

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器 f	处理数据	写寄存器 f

例：　　BSF　　FLAG_REG, 7, 1

执行指令前，

FLAG_REG　=　0Ah

执行指令后，

FLAG_REG　=　8Ah

BTFSC 对 f 做位测试，若为 0 则跳过

格式： BTFSC f,b{,a}

操作数： 0≤f≤255
0≤b≤7
a∈[0,1]

操作： 如果(f<b>)=0，跳过

受影响的标志位： 无

编码：

1011	bbba	ffff	ffff

描述： 如果 f 寄存器的 b 位为 0，跳过下一条指令。如果 b 位为 0，在取下一条指令时，放弃执行当前指令，取而代之的是执行一条 NOP 指令。因此，该指令需要 2 个指令周期。
如果 a=0，选择当前的访问存储区。如果 a=1，用 BSR 来选择 GPR 存储区(默认)。
如果 a=0 且扩展指令集可用，只要 f≤95(5Fh)，这条指令就会按立即数偏移量寻址模式执行。要获取更多的技术细节，请参阅 H.2.3 节。

字长： 1

周期： 1(2)
注意：如果跳过且紧跟的是双字指令，需要占用 3 个指令周期。

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器 f	处理数据	无操作

如果跳过：

Q1	Q2	Q3	Q4
无操作	无操作	无操作	无操作

如果跳过，并紧接一条双字指令：

Q1	Q2	Q3	Q4
无操作	无操作	无操作	无操作
无操作	无操作	无操作	无操作

例：

```
HERE BTFSC FLAG, 1, 0
FALSE:
TRUE:
```

执行指令前，
PC = 地址(HERE)
执行指令后，
如果 FLAG<1> = 0
PC = 地址(TRUE)
如果 FLAG<1> = 1
PC = 地址(FALSE)

BTFSS 对 f 做位测试，若为 1 则跳过

格式： BTFSS f,b{,a}

操作数： 0≤f≤255
0≤b<7
a∈[0,1]

操作： 如果(f<b>)=1，跳过。

受影响的标志位： 无

编码：

1010	bbba	ffff	ffff

描述： 如果 f 寄存器的 b 位为 1，跳过下一条指令。如果 b 位为 0，在取下一条指令时，放弃执行当前指令，取而代之的是执行一条 NOP 指令。因此，该指令需要 2 个指令周期。
如果 a=0，选择当前的访问存储区。如果 a=1，用 BSR 来选择 GPR 存储区(默认)。
如果 a=0 且扩展指令集可用，只要 f≤95(5Fh)，这条指令就会按立即数偏移量寻址模式执行。要获取更多的技术细节，请参阅 H.2.3 节。

字长： 1

周期： 1(2)
注意：如果跳过且紧跟的是双字指令，需要占用 3 个指令周期。

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器 f	处理数据	无操作

如果跳过：

Q1	Q2	Q3	Q4
无操作	无操作	无操作	无操作

如果跳过，并紧接一条双字指令：

Q1	Q2	Q3	Q4
无操作	无操作	无操作	无操作
无操作	无操作	无操作	无操作

例：

```
HERE BTFSS FLAG, 1, 0
FALSE:
TRUE:
```

执行指令前，
PC = 地址(HERE)
执行指令后，
如果 FLAG<1> = 0
PC = 地址(FALSE)
如果 FLAG<1> = 1
PC = 地址(TRUE)

BTG　　对f寄存器的位取反

格式：BTG f,b{,a}

操作数：0≤f≤255
0≤b<7
a∈[0,1]

操作：$\overline{(f<b>)}$→ f<b>

受影响的标志位：无

编码：

0111	bbba	ffff	ffff

描述：将f定位的数据寄存器的b位取反。如果a=0，选择当前的访问存储区。如果a=1，用BSR来选择GPR存储区（默认）。
如果a=0且扩展指令集可用，只要f≤95(5Fh)，这条指令就会按立即数偏移量寻址模式执行。要获取更多的技术细节，请参阅H.2.3节。

字长：1

周期：1

Q周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器f	处理数据	写寄存器f

例：　BTG　PORTC, 4, 0

执行指令前，
PORTC　=　0111 0101 [75h]
执行指令后，
PORTC　=　0110 0101 [65h]

BOV　　溢出则跳转

格式：BOV n

操作数：−128≤n≤127

操作：如果溢出标志位=1，
(PC)+2+2n→ PC

受影响的标志位：无

编码：

1110	0100	nnnn	nnnn

描述：如果溢出标志位为1，程序跳转。
二进制补码2n加到程序计数器上。因为PC在读取下一条指令时已经增加，新的地址为PC+2+2n。这条指令的执行需要2个指令周期。

字长：1

周期：1(2)

Q周期活动：

如果跳转：

Q1	Q2	Q3	Q4
译码	读立即数n	处理数据	写PC
无操作	无操作	无操作	无操作

如果不跳转：

Q1	Q2	Q3	Q4
译码	读立即数n	处理数据	无操作

例：　HERE BOV Jump

执行指令前，
PC　=　地址(HERE)
执行指令后，
如果溢出标志位　=　1
PC　=　地址(Jump)
如果溢出标志位　=　0
PC　=　地址(HERE+ 2)

BZ 为 0 则跳转

格式： BZ n

操作数： $-128 \leqslant n \leqslant 127$

操作： 如果零标志位=1，
(PC)+2+2n→ PC

受影响的标志位： 无

编码：

1110	0000	nnnn	nnnn

描述： 如果零标志位为 1，程序跳转。二进制补码 2n 加到程序计数器上。因为 PC 在读取下一条指令时已经增加，新的地址为 PC+2+2n。这条指令的执行需要 2 个指令周期。

字长： 1

周期： 1(2)

Q 周期活动：

如果跳转：

Q1	Q2	Q3	Q4
译码	读立即数 n	处理数据	写 PC
无操作	无操作	无操作	无操作

如果不跳转：

Q1	Q2	Q3	Q4
译码	读立即数 n	处理数据	无操作

例： HERE BZ Jump

执行指令前，
PC = 地址(HERE)

执行指令后，
如果零标志位 = 1
PC = 地址(Jump)
如果零标志位 = 0
PC = 地址(HERE+ 2)

CALL 调用子例程

格式： CALL k{,s}

操作数： $0 \leqslant k \leqslant 1\,048\,575$
$s \in [0,1]$

操作： (PC)+4 → TOS
k → PC<20:1>
如果 s=1，(W)→ WS，(Status)→ STATUSS，(BSR)→ BSRS

受影响的标志位： 无

编码：

1110	110s	$k_7$kkk	kkkk$_0$
1111	k_{19}kkk	kkkk	kkkk$_8$

描述： 调用整个 2 MB 存储范围内的子例程。首先，返回的地址(PC+4)被压栈。如果 s=1，W、BSR 和状态寄存器也会被压入各自对应的影子寄存器 WS、STATUSS 和 BSRS。如果 s=0，不会压入对应的影子寄存器(默认)，这时 k 的 20 位值装载到 PC<20:1>。CALL 指令是一个双周期指令。

字长： 2

周期： 2

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读立即数 k<7:0>	处理数据	读立即数 k<19:8>
无操作	无操作	无操作	无操作

例： HERE CALL THERE, 1

执行指令前，
PC = 地址(HERE)

执行指令后，
PC = 地址(THERE)
TOS = 地址(HERE+ 4)
WS = W
BSRS = BSR
STATUSS = Status

CLRF 清零 f 寄存器

格式： CLRF f{,a}

操作数： 0≤f≤255
a∈[0,1]

操作： 000h → f
1 → Z

受影响的标志位： Z

编码：

0110	101a	ffff	ffff

描述： 将目标寄存器清零。
如果 a=0，选择当前的访问存储区。
如果 a=1，用 BSR 来选择 GPR 存储区（默认）。
如果 a=0 且扩展指令集可用，只要 f≤95(5Fh)，这条指令就会按立即数偏移量寻址模式执行。要获取更多的技术细节，请参阅 H.2.3 节。

字长： 1

周期： 1

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器 f	处理数据	写寄存器 f

例： CLRF FLAG_REG, 1

执行指令前，
FLAG_REG = 5Ah
执行指令后，
FLAG_REG = 00h

CLRWDT 清零监视定时器

格式： CLRWDT

操作数： 无

操作： 000h → WDT
000h → WDT 后分频器
1 → $\overline{TO}$
1 → $\overline{PD}$

受影响的标志位： $\overline{TO}$，$\overline{PD}$

编码：

0000	0000	0000	0100

描述： CLRWDT 指令将复位监视定时器。它同时也对后分频器进行复位。状态位 $\overline{TO}$ 和 $\overline{PD}$ 都被置 1。

字长： 1

周期： 1

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器 f	处理数据	无操作

例： CLRWDT

执行指令前，WDT 计数器 = ?
执行指令后，
WDT 计数器 = 00h
WDT 后分频器 = 0
$\overline{TO}$ = 1
$\overline{PD}$ = 1

COMF 对 f 寄存器的内容取反

格式： COMF f{,d{,a}}

操作数： 0≤f≤255
d∈[0,1]
a∈[0,1]

操作： $(\bar{f})$ → 目的寄存器

受影响的标志位： N, Z

编码：

0001	11da	ffff	ffff

描述： 将 f 寄存器内容取反。如果 d=0,结果放入 W。如果 d=1,结果放入 f 寄存器(默认)。

如果 a=0,选择当前的访问存储区。如果 a=1,用 BSR 选择 GPR 存储区(默认)。

如果 a=0 且扩展指令集可用,只要 f≤95(5Fh),这条指令就会按立即数偏移量寻址模式执行。要获取更多的技术细节,请参阅 H.2.3 节。

字长： 1

周期： 1

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器 f	处理数据	写目的寄存器

例：

```
COMF    REG, 0, 0
```

执行指令前，
REG = 13h
执行指令后，
REG = 13h
W = ECh

CPFSEQ 比较 f 和 WREG,相等则跳过

格式： CPFSEQ f{,a}

操作数： 0≤f≤255
a∈[0,1]

操作： (f) －(W),如果相等,跳过。(无符号数比较)

受影响的标志位： 无

编码：

0110	001a	ffff	ffff

描述： 用无符号数的减法,比较 W 的内容和数据寄存器的内容。如果 f=W,放弃已经取得的指令,取而代之的是执行一条 NOP 指令,为 2 个指令周期。

如果 a=0,选择当前的访问存储区。如果 a=1,用 BSR 来选择 GPR 存储区(默认)。

如果 a=0 且扩展指令集可用,只要 f≤95(5Fh),这条指令就会按立即数偏移量寻址模式执行。要获取更多的技术细节,请参阅 H.2.3 节。

字长： 1

周期： 1(2)
注意:如果跳过且紧跟的是双字指令,需要占用 3 个指令周期。

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器 f	处理数据	无操作

如果跳过：

Q1	Q2	Q3	Q4
无操作	无操作	无操作	无操作

如果跳过,并紧接一条双字指令：

Q1	Q2	Q3	Q4
无操作	无操作	无操作	无操作
无操作	无操作	无操作	无操作

例：

```
HERE CPFSEQ REG, 0
NEQUAL:
EQUAL:
```

执行指令前，
PC 地址 = HERE
W = ?
REG = ?
执行指令后，
如果 REG = W
PC = 地址(EQUAL)
如果 REG 不等于 W
PC = 地址(NEQUAL)

CPFSGT　比较f和W，若f>W则跳过

格式：　CPFSGT f{,a}

操作数：　0≤f≤255
a∈[0,1]

操作：　(f) −(W)，如果大于，跳过。(无符号数比较)

受影响的标志位：　无

编码：

0110	010a	ffff	ffff

描述：　用无符号数的减法，比较W的内容和数据寄存器的内容。如果f大于W，放弃已经取得的指令，取而代之的是执行一条NOP指令，为2个指令周期。
如果a=0，选择当前的访问存储区。如果a=1，用BSR来选择GPR存储区(默认)。
如果a=0且扩展指令集可用，只要f≤95(5Fh)，这条指令就会按立即数偏移量寻址模式执行。要获取更多的技术细节，请参阅H.2.3节。

字长：　1

周期：　1(2)

注意：如果跳过且紧跟的是双字指令，需要占用3个指令周期。

Q周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器f	处理数据	无操作

如果跳过：

Q1	Q2	Q3	Q4
无操作	无操作	无操作	无操作

如果跳过，并紧接一条双字指令：

Q1	Q2	Q3	Q4
无操作	无操作	无操作	无操作
无操作	无操作	无操作	无操作

例：
```
HERE     CPFSGT REG, 0
NGREATER   :
GREATER    :
```

执行指令前，
PC　=　地址(HERE)
W=?
执行指令后，
如果REG大于W，
PC　=　地址(GREATER)
如果REG不大于W，
PC　=　地址(NGREATER)

CPFSLT　比较f和W，f<W则跳过

格式：　CPFSLT f{,a}

操作数：　0≤f≤255
a∈[0,1]

操作：　(f) −(W)，如果小于，跳过。(无符号数比较)

受影响的标志位：　无

编码：

0110	000a	ffff	ffff

描述：　用无符号数的减法，比较W的内容和数据寄存器的内容。如果f小于W，放弃已经取得的指令，取而代之的是执行一条NOP指令，为2个指令周期。
如果a=0，选择当前的访问存储区。如果a=1，用BSR选择GPR存储区(默认)。

字长：　1

周期：　1(2)
注意：如果跳过且紧跟的是双字指令，需要占用3个指令周期。

Q周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器f	处理数据	无操作

如果跳过：

Q1	Q2	Q3	Q4
无操作	无操作	无操作	无操作

如果跳过，并紧接一条双字指令：

Q1	Q2	Q3	Q4
无操作	无操作	无操作	无操作
无操作	无操作	无操作	无操作

例：
```
HERE   CPFSLT REG, 1
NLESS    :
LESS     :
```

执行指令前，
PC　=　地址(HERE)
W　=　?
执行指令后，
如果REG小于W，
PC　=　地址(LESS)
如果REG不小于W，
PC　=　地址(NLESS)

DAW　　W 寄存器十进制校正

格式：　DAW

操作数：　无

操作：　如果 W<3:0>大于 9 或者 DC=1，那么 W<3:0>+6 → W<3:0>；否则，W<3:0> → W<3:0>。
如果 W<7:4>大于 9 或者 C=1，那么 W<7:4>+6 → W<7:4>，C=1；否则，W<7:4> → W<7:4>。

受影响的标志位：　C

编码：

0000	0000	0000	0111

描述：　DAW 指令调整 W 寄存器中的 8 位数据，将两个 BCD 格式的变量调整，产生一个正确的 BCD 结果。

字长：　1

周期：　1

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器 W	处理数据	写寄存器 W

例 1：DAW

执行指令前，
W = A5h
C = 0
DC = 0

执行指令后，
W = 05h
C = 1
DC = 0

例 2：DAW

执行指令前，
W = CEh
C = 0
DC = 0

执行指令后，
W = 34h
C = 1
DC = 0

DECF　　对 f 寄存器的内容做减 1 操作

格式：　DECF f{,d{,a}}

操作数：　0≤f≤255
d∈[0,1]
a∈[0,1]

操作：　(f)−1 → 目的地址

受影响的标志位：　C,DC,N,OV,Z

编码：

0000	01da	ffff	ffff

描述：　将 f 寄存器的内容减 1。如果 d=0，结果放入 W。如果 d=1，结果放入 f 寄存器(默认)。
如果 a=0，选择当前的访问存储区。如果 a=1，用 BSR 选择 GPR 存储区(默认)。
如果 a=0 且扩展指令集可用，只要 f≤95(5Fh)，这条指令就会按立即数偏移量寻址模式执行。要获取更多的技术细节，请参阅 H.2.3 节。

字长：　1

周期：　1

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器 f	处理数据	写目的寄存器

例：　DECF CNT, 1, 0

执行指令前，
CNT = 01h
Z = 0

执行指令后，
CNT = 00h
Z = 1

DECFSZ 对 f 寄存器的内容做减 1 操作，若为 0 则跳过

格式： DECFSZ f{,d{,a}}

操作数： 0≤f≤255
d∈[0,1]
a∈[0,1]

操作： (f)−1 → 目的地址
若结果等于零，跳过。

受影响的标志位： 无

编码：

0010	11da	ffff	ffff

描述： 对 f 寄存器的内容做减 1 运算。如果 d=0，结果放入 W。如果 d=1，结果放入 f 寄存器(默认)。如果结果为 0，丢弃当前已取得的指令，取而代之的是执行一条 NOP 指令，使其成为双周期指令。
如果 a=0，选择当前的访问存储区。如果 a=1，用 BSR 选择 GPR 存储区(默认)。
如果 a=0 且扩展指令集可用，只要 f≤95(5Fh)，这条指令就会按立即数偏移量寻址模式执行。要获取更多的技术细节，请参阅 H.2.3 节。

字长： 1

周期： 1(2)
注意：如果跳过且紧跟的是双字指令，需要占用 3 个指令周期。

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器 f	处理数据	写目的寄存器

如果跳过：

Q1	Q2	Q3	Q4
无操作	无操作	无操作	无操作

如果跳过并紧接一条双字指令：

Q1	Q2	Q3	Q4
无操作	无操作	无操作	无操作
无操作	无操作	无操作	无操作

例：
```
HERE      DECFSZ   CNT, 1, 1
          GOTO     LOOP
CONTINUE
```

执行指令前，
PC = 地址(HERE)
执行指令后，
CNT = CNT−1
如果 CNT = 0
PC = 地址(CONTINUE)
如果 CNT 不等于 0
PC = 地址(HERE+ 2)

DCFSNZ 对 f 寄存器的内容做减 1 操作，若为非 0 则跳过

格式： DCFSNZ f{,d{,a}}

操作数： 0≤f≤255
d∈[0,1]
a∈[0,1]

操作： (f)−1 → 目的地址
若结果不等于零，则跳过

受影响的标志位： 无

编码：

0100	11da	ffff	ffff

描述： 对 f 寄存器的内容做减 1 运算。如果 d=0，结果放入 W。如果 d=1，结果放入 f 寄存器(默认)。如果结果不为 0，丢弃当前已取得的指令，取而代之的是执行一条 NOP 指令，使其成为双周期指令。
如果 a=0，选择当前的访问存储区。如果 a=1，用 BSR 选择 GPR 存储区(默认)。
如果 a=0 且扩展指令集可用，只要 f≤95(5Fh)，这条指令就会按立即数偏移量寻址模式执行。要获取更多的技术细节，请参阅 H.2.3 节。

字长： 1

周期： 1(2)
注意：如果跳过且紧跟的是双字指令，需要占用 3 个指令周期。

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器 f	处理数据	写目的寄存器

如果跳过：

Q1	Q2	Q3	Q4
无操作	无操作	无操作	无操作

如果跳过并紧接一条双字指令：

Q1	Q2	Q3	Q4
无操作	无操作	无操作	无操作
无操作	无操作	无操作	无操作

例：
```
HERE   DCFSNZ   TEMP, 1, 0
ZERO:
NZERO:
```

执行指令前，
TEMP = ?
执行指令后，
TEMP = TEMP −1
如果 TEMP = 0
PC = 地址(ZERO)
如果 TEMP 不等于 0
PC = 地址(NZERO)

GOTO 无条件跳转

格式： GOTO k

操作数： 0≤k≤1 048 575

操作： k →PC<20:1>

受影响的标志位： 无。

编码：

1110	1111	$k_7$kkk	kkkk$_0$
1111	k_{19}kkk	kkkk	kkkk$_8$

描述： GOTO 指令允许在整个 2MB 存储范围内的无条件跳转。这时，k 的 20 位值装入 PC<20:1>。GOTO 是一条双周期指令。

字长： 2

周期： 2

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读立即数 k<7:0>	无操作	读立即数 k<19:8>，写入 PC
无操作	无操作	无操作	无操作

例： GOTO THERE

执行指令后，

PC = 地址(THERE)

INCF 对 f 寄存器的内容做加 1 操作

格式： INCF f{,d{,a}}

操作数： 0≤f≤255

d∈[0,1]

a∈[0,1]

操作： (f)+1→目的地址

受影响的标志位： C,DC,N,OV,Z

编码：

0010	10da	ffff	ffff

描述： 对 f 寄存器的内容进行加 1 操作。如果 d=0，结果放入 W。如果 d=1，结果放入 f 寄存器(默认)。

如果 a=0，选择当前的访问存储区。如果 a=1，用 BSR 选择 GPR 存储区(默认)。

如果 a=0 且扩展指令集可用，只要 f≤95(5Fh)，这条指令就会按立即数偏移量寻址模式执行。要获取更多的技术细节，请参阅 H.2.3 节。

字长： 1

周期： 1

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器 f	处理数据	写目的寄存器

例： INCF CNT, 1, 0

执行指令前，

CNT = FFh
Z = 0
C = ?
DC = ?

执行指令后，

CNT = 00h
Z = 1
C = 1
DC = 1

INCFSZ 对 f 寄存器的内容加 1，若为 0 则跳过

格式： INCFSZ f{,d{,a}}

操作数： 0≤f≤255
d∈[0,1]
a∈[0,1]

操作： (f)+1→目的地址。若结果等于 0，则跳过。

受影响的标志位： 无。

编码：

0011	11da	ffff	ffff

描述： 对 f 寄存器的内容加 1。如果 d=0，结果放入 W。如果 d=1，结果放入 f 寄存器(默认)。
如果结果为 0，丢弃当前已取得的指令，取而代之的是执行一条 NOP 指令，使其成为双周期指令。
如果 a=0，选择当前的访问存储区。如果 a=1，用 BSR 选择 GPR 存储区(默认)。
如果 a=0 且扩展指令集可用，只要 f≤95 (5Fh)，这条指令就会按立即数偏移量寻址模式执行。要获取更多的技术细节，请参阅 H.2.3 节。

字长： 1

周期： 1(2)

注意：如果跳过且紧跟的是双字指令，需要占用 3 个指令周期。

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器 f	处理数据	写目的寄存器

如果跳过：

Q1	Q2	Q3	Q4
无操作	无操作	无操作	无操作

如果跳过并紧接一条双字指令：

Q1	Q2	Q3	Q4
无操作	无操作	无操作	无操作
无操作	无操作	无操作	无操作

例：

```
HERE    INCFSZ CNT,1,0
NZERO:
ZERO :
```

执行指令前，
PC = 地址(HERE)
执行指令后，
CNT = CNT+1
如果 CNT = 0
PC = 地址(ZERO)
如果 CNT 不等于 0，
PC = 地址(NZERO)

INFSNZ 对 f 寄存器的内容加 1，若不为 0 则跳过

格式： INFSNZ f{,d{,a}}

操作数： 0≤f≤255
d∈[0,1]
a∈[0,1]

操作： (f)+1→目的地址。若结果不等于 0，则跳过。

受影响的标志位： 无。

编码：

0100	10da	ffff	ffff

描述： 对 f 寄存器的内容加 1。如果 d=0，结果放入 W。如果 d=1，结果放入 f 寄存器(默认)。
如果结果不为 0，丢弃当前已取得的指令，取而代之的是执行一条 NOP 指令，使其成为双周期指令。
如果 a=0，选择当前的访问存储区。如果 a=1，用 BSR 选择 GPR 存储区(默认)。
如果 a=0 且扩展指令集可用，只要 f≤95 (5Fh)，这条指令就会按立即数偏移量寻址模式执行。要获取更多的技术细节，请参阅 H.2.3 节。

字长： 1

周期： 1(2)

注意：如果跳过且紧跟的是双字指令，需要占用 3 个指令周期。

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器 f	处理数据	写目的寄存器

如果跳过：

Q1	Q2	Q3	Q4
无操作	无操作	无操作	无操作

如果跳过并紧接一条双字指令：

Q1	Q2	Q3	Q4
无操作	无操作	无操作	无操作
无操作	无操作	无操作	无操作

例：

```
HERE    INFSNZ REG,1,0
ZERO
NZERO
```

执行指令前，
PC = 地址(HERE)
执行指令后，
CNT = CNT +1
如果 CNT = 0
PC = 地址(NZERO)
如果 CNT 不等于 0
PC = 地址(ZERO)

IORLW 用立即数同 W 的内容做逻辑或运算

格式： IORLW k

操作数： 0≤k≤255

操作： (W)逻辑或 k→W

受影响的标志位： N,Z

编码：

0000	1001	kkkk	kkkk

描述： 将 W 的内容和 8 位立即数进行逻辑或运算，结果放入 W。

字长： 1

周期： 1

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读立即数 k	处理数据	写 W 寄存器

例： IORLW 35h

执行指令前，

W = 9Ah

执行指令后，

W = BFh

IORWF 将 W 的内容和 f 的内容进行逻辑或运算

格式： IORWF f{,d{,a}}

操作数： 0≤f≤255

d∈[0,1]

a∈[0,1]

操作： (W)逻辑异或 f→目的地址

受影响的标志位： N,Z

编码：

0001	00da	ffff	ffff

描述： 将 W 的内容和 f 寄存器的内容做逻辑或运算。如果 d=0，结果放入 W。如果 d=1，结果放入 f 寄存器(默认)。

如果 a=0，选择当前的访问存储区。如果 a=1，用 BSR 来选择 GPR 存储区(默认)。

如果 a=0 且扩展指令集可用，只要 f≤95 (5Fh)，这条指令就会按立即数偏移量寻址模式执行。要获取更多的技术细节，请参阅 H.2.3 节。

字长： 1

周期： 1

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器 f	处理数据	写目的寄存器

例： IORWF RESULT, 0, 1

执行指令前，

RESULT = 13h

W = 91h

执行指令后，

RESULT = 13h

W = 93h

LFSR　　装载 FSR

格式：　LFSR　f,k

操作数：　0≤f≤2
　0≤k≤4095

操作：　k→FSRf

受影响的标志位：　无。

编码：

1110	1110	00ff	k_{11}kkk
1111	0000	$k_7$kkk	kkkk

描述：　将 12 位的立即数 k 写入 f 指定的 FSR 寄存器。

字长：　2

周期：　2

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读立即数 k 的高位(MSB)	处理数据	写立即数 k 的 MSB 到 FSRfH
译码	读立即数 k 的低位(LSB)	处理数据	写立即数 k 的 LSB 到 FSRfL

例：　LFSR　2, 3ABh

执行指令后，

FSR2H　=　03h

FSR2L　=　ABh

MOVF　　传送 f 寄存器的内容

格式：　MOVF　f{,d{,a}}

操作数：　0≤k≤255
　d∈[0,1]
　a∈[0,1]

操作：　(f)→ 目的地址

受影响的标志位：　N,Z

编码：

0101	00da	ffff	ffff

描述：　根据 d 的值将 f 寄存器的内容传送到目的地址。如果 d=0，结果放入 W。如果 d=1，结果放入 f 寄存器(默认)。f 可以是 256B 存储区里的任何位置。

如果 a=0，选择当前的访问存储区。如果 a=1，用 BSR 选择 GPR 存储区(默认)。

如果 a=0 且扩展指令集可用，只要 f≤95(5Fh)，这条指令就会按立即数偏移量寻址模式执行。要获取更多的技术细节，请参阅 H.2.3 节。

字长：　1

周期：　1

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器 f	处理数据	写 W 寄存器

例：　MOVF　REG, 0, 0

执行指令前，

REG　=　22h

W　=　FFh

执行指令后，

REG　=　22h

W　=　22h

MOVFF f 移动到 f

格式： MOVFF f_s，f_d

操作数： $0 \leqslant f_s \leqslant 4095$

$0 \leqslant f_d \leqslant 4095$

操作： $(f_s) \rightarrow f_d$

受影响的标志位： 无。

编码：

1100	ffff	ffff	$ffff_s$
1111	ffff	ffff	$ffff_d$

描述： 将源寄存器 f_s 的内容传送到目的寄存器 f_d 中。源寄存器可以位于任何一个容量为 4096B 的存储空间（000h～FFFh）内，目的寄存器也可以是 000h～FFFh 内任何位置的寄存器。

源寄存器和目的寄存器其中必有一个有 W 寄存器的功能（SFR）。

MOVFF 指令对于传输数据存储器的数据到外围模块寄存器是非常有效的，如传送缓冲区或 I/O 端口，但 MOVFF 指令不能用 PCL、TOSU、TOSH 或者 TOSL 作为目的寄存器。

字长： 2

周期： 2(3)

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器 f(src)	处理数据	无操作
译码	无操作，无哑元可读	处理数据	写寄存器 f（目的寄存器）

例： MOVFF REG1，REG2

执行指令前，

REG1 = 33h

REG2 = 11h

执行指令后，

REG1 = 33h

REG2 = 33h

MOVLB 将立即数传送到 BSR

格式： MOVLB k

操作数： $0 \leqslant k \leqslant 255$

操作： $k \rightarrow BSR$

受影响的标志位： 无。

编码：

0000	0001	kkkk	kkkk

描述： 将 8 位的立即数传送入 BSR 寄存器。BSR<7：4>的值总是保持为 0，与 k_7：k_4的值无关。

字长： 1

周期： 1

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读立即数 k	处理数据	写立即数 k 到 BSR

例： MOVLB 5

执行指令前，

BSR 寄存器 = 02h

执行指令后，

BSR 寄存器 = 05h

MOVLW 立即数送入 W

格式： MOVLW k

操作数： 0≤k≤255

操作： k → W

受影响的标志位： 无。

编码：

0000	1110	kkkk	kkkk

描述： 将 8 位的立即数传送到 W 寄存器中

字长： 1

周期： 1

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读立即数 k	处理数据	写 W 寄存器

例： MOVLW 5Ah

执行指令后，

W = 5Ah

MOVWF 将 W 的内容传送至 f 寄存器

格式： MOVWF f{,a}

操作数： 0≤f≤255

a∈[0,1]

操作： (W) → f

受影响的标志位： 无。

编码：

0110	111a	ffff	ffff

描述： 将数据从 W 寄存器传送到 f 寄存器。F 寄存器的位置可处在 256B RAM 存储区的任何地方。

如果 a=0，选择当前的访问存储区。

如果 a=1，用 BSR 选择 GPR 存储区（默认）。

如果 a=0 且扩展指令集可用，只要 f≤95(5Fh)，这条指令就会按立即数偏移量寻址模式执行。要获取更多的技术细节，请参阅 H.2.3 节。

字长： 1

周期： 1

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器 f	处理数据	写寄存器 f

例： MOVWF REG,0

执行指令前，

W = 4Fh

REG = FFh

执行指令后，

W = 4Fh

REG = 4Fh

MULLW 立即数和 W 相乘

格式： MULLW k

操作数： 0≤k≤255

操作： (W)×k →PRODH:PRODL

受影响的标志位： 无。

编码：

0000	1101	kkkk	kkkk

描述： 将 W 寄存器的内容乘以 8 位的立即数(无符号数的乘法)。所得的 16 位结果放入 PRODH:PRODL 寄存器，其中 PRODH 存放高字节。
W 寄存器的内容保持不变。
不影响任何状态标志位。
注意，执行这条指令不会产生溢出标志位和进位标志位。可能会出现零标志位，但是不会去检测。

字长： 1

周期： 1

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读立即数 k	处理数据	写寄存器 PRODH:PRODL

例： MULLW 0C4h

执行指令前，
W = E2h
PRODH = ?
PRODL = ?

执行指令后，
W = E2h
PRODH = ADh
PRODL = 08h

MULWF 将 W 寄存器的内容和 f 寄存器的内容相乘

格式： MULWF f{,a}

操作数： 0≤f≤255
a∈[0,1]

操作： (W)×(f) →PRODH:PRODL

受影响的标志位： 无。

编码：

0000	001a	ffff	ffff

描述： 将 W 寄存器的内容和 f 寄存器的内容进行无符号数的乘法运算。所得的 16 位结果放入 PRODH:PRODL 寄存器对，其中 PRODH 放高字节。
W 寄存器的内容保持不变。
不影响任何状态标志位。
注意，执行这条指令不会产生溢出标志位和进位标志位。可能会出现零标志位，但是不会去检测。
如果 a=0，选择当前的访问存储区。
如果 a=1，用 BSR 选择 GPR 存储区(默认)。
如果 a=0 且扩展指令集可用，只要 f≤95(5Fh)，这条指令就会按立即数偏移量寻址模式执行。要获取更多的技术细节，请参阅 H.2.3 节。

字长： 1

周期： 1

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器 f	处理数据	写寄存器 PRODH:PRODL

例： MULWF REG,1

执行指令前，
W = C4h
REG = B5h
PRODH = ?
PRODL = ?

执行指令后，
W = C4h
REG = B5h
PRODH = 8Ah
PRODL = 94h

NEGF 对 f 寄存器的内容做取补运算

格式： NEGF f{,a}

操作数： 0≤f≤255
a∈[0,1]

操作： $(\bar{f})+1 \rightarrow f$

受影响的标志位： N,OV,C,DC,Z

编码：

0110	110a	ffff	ffff

描述： 将 f 寄存器的内容取补。结果放入数据存储器 f。
如果 a=0，选择当前的访问存储区。
如果 a=1，用 BSR 选择 GPR 存储区（默认）。
如果 a=0 且扩展指令集可用，只要 f≤95(5Fh)，这条指令就会按立即数偏移量寻址模式执行。要获取更多的技术细节，请参阅 H.2.3 节。

字长： 1

周期： 1

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器 f	处理数据	写寄存器 f

例： NEGF REG, 1

执行指令前，
REG = 0011 1010 [3Ah]

执行指令后，
REG = 1100 0110 [C6h]

NOP 空操作

格式： NOP

操作数： 无。

操作： 无操作。

受影响的标志位： 无。

编码：

0000	0000	0000	0000
1111	xxxx	xxxx	xxxx

描述： 无操作。

字长： 1

周期： 1

Q 周期活动：

Q1	Q2	Q3	Q4
译码	无操作	无操作	无操作

例： 无。

POP 返回栈出栈操作

格式： POP

操作数： 无。

操作： (TOS)→ 位桶式寄存器(bit bucket)

受影响的标志位： 无。

编码：

0000	0000	0000	0110

描述： 返回栈栈顶值被弹出并被丢弃，栈顶(TOS)值为先前一个压栈的值。该指令能够有效地帮助设计人员管理栈，以便生成软件栈。

字长： 1

周期： 1

Q 周期活动：

Q1	Q2	Q3	Q4
译码	无操作	弹出栈顶(TOS)值	不操作

例： POP
GOTO NEW

执行指令前：
TOS = 0031hA2h
栈(低一层压栈值) = 014332h

执行指令后：
TOS = 014332h
PC = NEW(新的值)

PUSH 返回栈入栈操作

格式： PUSH

操作数： 无。

操作： (PC+2)→ TOS

受影响的标志位： 无。

编码：

0000	0000	0000	0101

描述： PC+2 被压入返回栈的栈顶。栈顶(TOS)先前被压入栈的下一层。该指令允许用户通过修改 TOS 实现软件栈。

字长： 1

周期： 1

Q 周期活动：

Q1	Q2	Q3	Q4
译码	将(PC=2)压栈	无操作	无操作

例： PUSH

执行指令前：
TOS = 345Ah
PC = 0124h

执行指令后：
PC = 0126h
TOS = 0126h
栈(下一层栈值) = 345Ah

RCALL　　相对调用操作

格式：　RCALL n

操作数：　$-1024 \leq n \leq 1023$

操作：　(PC)+2→TOS
(PC)+2+2n → PC

受影响的标志位：　无。

编码：

1101	1nnn	nnnn	nnnn

描述：　相对子例程调用，可在当前位置的1KB空间区域内调用并执行子例程。首先，返回地址(PC+2)被压栈。然后，二进制补码2n被加到PC上。因为PC在读取下一条指令时已经增加，新的地址为PC+2+2n。这条指令是双周期指令。

字长：　1

周期：　2

Q周期活动：

Q1	Q2	Q3	Q4
译码	读立即数n，将PC压栈	处理数据	写PC
无操作	无操作	无操作	无操作

例：　HERE RCALL Jump

执行指令前，
　PC　=　地址(HERE)

执行指令后，
　PC　=　地址(Jump)
　TOS　=　地址(HERE+ 2)

RESET　　复位操作

格式：　RESET

操作数：　无。

操作：　通过$\overline{MCLR}$复位引脚复位所有的寄存器和标志位。

受影响的标志位：　所有的标志位。

编码：

0000	0000	1111	1111

描述：　该指令提供了一种软件操作$\overline{MCLR}$复位的方法。

字长：　1

周期：　1

Q周期活动：

Q1	Q2	Q3	Q4
译码	开始复位	无操作	无操作

例：　RESET

执行指令后，
　寄存器=复位值
　标志位* =复位值

RETFIE 中断返回

格式： RETFIE {s}

操作数： s∈[0,1]

操作： (TOS)→ PC
1 → GIE/GIEH 或者 PEIE/GIEL
如果 s=1,那么
(WS)→ W
(STATUSS)→ Status
(BSRS)→ BSR
PCLATU、PCLATH 保持不变

受影响的标志位： GIE/GIEH,PEIE/GIEL

编码：

0000	0000	0001	000s

描述： 中断返回。进行出栈操作,栈顶值(TOS)送入 PC。中断由优先权高或低的全局中断发起。如果 s=1,影子寄存器 WS、STATUSS 和 BSRS 的内容将被送入到对应的寄存器 W、STATUS 和 BSR。如果 s=0,这些寄存器将不更新(默认)。

字长： 1

周期： 2

Q 周期活动：

Q1	Q2	Q3	Q4
译码	无操作	无操作	从栈顶弹出 PC,置位 GIEH 或 GIEL
无操作	无操作	无操作	无操作

例： RETFIE 1

执行指令后，

PC = TOS
W = WS
BSR = BSRS
STATUS = STATUSS
GIE/GIEH,PEIE/GIEL = 1

RETLW 返回立即数到 W 寄存器

格式： RETLW k

操作数： 0≤k≤255

操作： k → W
(TOS)→ PC
PCLATU、PCLATH 保持不变

受影响的标志位： 无。

编码：

0000	1100	kkkk	kkkk

描述： 将 8 位的立即数送入 W 寄存器。程序计数器从栈顶处加载(返回地址)。高地址锁存器 PCLATH 保持不变。

字长： 1

周期： 2

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读立即数 k	处理数据	从栈顶弹出 PC,写入 W 寄存器
无操作	无操作	无操作	无操作

例：

```
      CALL   TABLE  ; W contains table
                    ; offset value
                    ; W now has
                    ; table value
      :
TABLE
      ADDWF  PCL    ; W = offset
      RETLW  k0     ; Begin table
      RETLW  k1     ;
      :
      :
      RETLW  kn     ; End of table
```

执行指令前，

W = 07h

执行指令后，

W = kn 的值

RETURN　子例程返回

格式：　RETURN {s}

操作数：　s∈[0,1]

操作：　(TOS)→ PC
如果 s=1，那么：
(WS)→ W
(STATUSS)→ Status
(BSRS)→ BSR
PCLATU、PCLATH 保持不变

受影响的标志位：　无。

编码：

0000	0000	0001	001s

描述：　从子例程返回。进行出栈操作，将栈顶值(TOS)装入程序计数器。如果 s=1，影子寄存器 WS、STATUSS 和 BSRS 的内容将被送入对应的寄存器 W、状态寄存器和 BSR。如果 s=0，不修改这些寄存器(默认)。

字长：　1

周期：　2

Q 周期活动：

Q1	Q2	Q3	Q4
译码	无操作	处理数据	从栈顶弹出 PC
无操作	无操作	无操作	无操作

例：　RETURN

执行指令后，
PC　=TOS

RLCF　带进/借位循环左移

格式：　RLCF f{,d{,a}}

操作数：　0≤f≤255
d∈[0,1]
a∈[0,1]

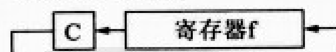
操作：　(f<n>)→ 目的地址<n+1>
(f<7>) → C
C → 目的地址<0>

受影响的标志位：　C,N,Z

编码：

0011	01da	ffff	ffff

描述：　将 f 寄存器的内容带进位/借位标志位循环左移 1 位。如果 d=0，结果放入 W。如果 d=1，结果放入 f 寄存器(默认)。
如果 a=0，选择当前的访问存储区。如果 a=1，用 BSR 选择 GPR 存储区(默认)。
如果 a=0 且扩展指令集可用，只要 f≤95 (5Fh)，这条指令就会按立即数偏移量寻址模式执行。要获取更多的技术细节，请参阅 H.2.3 节。



字长：　1

周期：　1

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器 f	处理数据	写目的地址

例：　RLCF REG, 0, 0

执行指令前，
REG　=　1110 0110
C　=　0

执行指令后，
REG　=　1110 0110
W　=　1100 1100
C　=　1

RLNCF 循环左移(不带进/借位)

格式： RLNCF f{,d{,a}}

操作数： 0≤f≤255
d∈[0,1]
a∈[0,1]

操作： (f<n>)→目的地址<n+1>
(f<7>)→目的地址<7>

受影响的标志位： N,Z

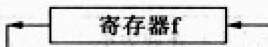
编码：

0100	01da	ffff	ffff

描述： 将f寄存器的内容循环左移1位。如果d=0,结果放入W。如果d=1,结果放入f寄存器(默认)。

如果a=0,选择当前的访问存储区。如果a=1,用BSR选择GPR存储区(默认)。

如果a=0且扩展指令集可用,只要f≤95(5Fh),这条指令就会按立即数偏移量寻址模式执行。要获取更多的技术细节,请参阅H.2.3节。



字长： 1

周期： 1

Q周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器 f	处理数据	写目的地址

例： RLNCF REG, 1, 0

执行指令前，
REG = 1010 1011

执行指令后，
REG = 0101 0111

RRCF 带进/借位循环右移

格式： RRCF f{,d{,a}}

操作数： 0≤f≤255
d∈[0,1]
a∈[0,1]

操作： (f<n>)→目的地址<n−1>
(f<0>)→C
C→目的地址<7>

受影响的标志位： C,N,Z

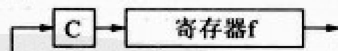
编码：

0011	00da	ffff	ffff

描述： 将f寄存器的内容带进位/借位标志位循环右移1位。如果d=0,结果放入W。如果d=1,结果放入f寄存器(默认)。

如果a=0,选择当前的访问存储区。如果a=1,用BSR选择GPR存储区(默认)。

如果a=0且扩展指令集可用,只要f≤95(5Fh),这条指令就会按立即数偏移量寻址模式执行。要获取更多的技术细节,请参阅H.2.3节。



字长： 1

周期： 1

Q周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器 f	处理数据	写目的地址

例： RRCF REG, 0, 0

执行指令前，
REG = 1110 0110
C = 0

执行指令后，
REG = 1110 0110
W = 0111 0011
C = 0

RRNCF　　循环右移(不带进/借位)

格式：　RRNCF f{,d{,a}}

操作数：　0≤f≤255
d∈[0,1]
a∈[0,1]

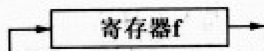
操作：　(f<n>)→ 目的地址<n−1>
(f<0>) →目的地址<7>

受影响的标志位：　N,Z

编码：

0100	00da	ffff	ffff

描述：　将 f 寄存器的内容循环右移 1 位。如果 d=0,结果放入 W。如果 d=1,结果放入 f 寄存器(默认)。
如果 a=0,选择当前的访问存储区。如果 a=1,用 BSR 选择 GPR 存储区(默认)。
如果 a=0 且扩展指令集可用,只要 f≤95(5Fh),这条指令就会按立即数偏移量寻址模式执行。要获取更多的技术细节,请参阅 H.2.3 节。



字长：　1

周期：　1

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器 f	处理数据	写目的地址

例 1：　RRNCF REG, 1, 0

执行指令前,REG　=　1101 0111
执行指令后,REG　=　1110 1011

例 2：　RRNCF REG, 0, 0

执行指令前,
W　=　?
REG　=　1101 0111
执行指令后,
W　=　1110 1011
REG　=　1101 0111

SETF　　将 f 置位为 1

格式：　SETF f{,a}

操作数：　0≤f≤255
a∈[0,1]

操作：　FFh → f

受影响的标志位：　无。

编码：

0110	100a	ffff	ffff

描述：　将指定寄存器的内容置为 FFH。
如果 a=0,选择当前的访问存储区。如果 a=1,用 BSR 选择 GPR 存储区(默认)。
如果 a=0 且扩展指令集可用,只要 f≤95(5Fh),这条指令就会按立即数偏移量寻址模式执行。要获取更多的技术细节,请参阅 H.2.3 节。

字长：　1

周期：　1

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器 f	处理数据	写寄存器 f

例：　SETF REG, 1

执行指令前,
REG　=　5Ah
执行指令后,
REG　=　FFh

SLEEP 进入待机模式

格式： SLEEP

操作数： 无

操作： 00h → WDT

0 → WDT 后分频器

1 → $\overline{TO}$

0 → $\overline{PD}$

受影响的标志位： $\overline{TO}$,$\overline{PD}$

编码：

0000	0000	0000	0011

描述： 掉电标志位($\overline{PD}$)被清零。时间溢出标志位$\overline{TO}$被置1。监视定时器和后分频器被清零。

主振荡器停止工作，处理器进入待机模式。

字长： 1

周期： 1

Q 周期活动：

Q1	Q2	Q3	Q4
译码	无操作	处理数据	进入待机模式

例： SLEEP

执行指令前，

$\overline{TO}$ = ?

$\overline{PD}$ = ?

执行指令后，

$\overline{TO}$ = 1*

$\overline{PD}$ = 0

* 如果让 WDT 唤醒系统，那么该位应被清零。

SUBFWB 借位减法

格式： SUBFWB f{,d{,a}}

操作数： 0≤f≤255

d∈[0,1]

a∈[0,1]

操作： (W)−(f)−($\overline{C}$)→目的地址

受影响的标志位： N,OV,C,DC,Z

编码：

0101	01da	ffff	ffff

描述： 将 W 寄存器的内容减去 f 寄存器的内容和借位(二进制补码形式)。如果 d=0,结果放入 W。如果 d=1,结果放入 f 寄存器(默认)。

如果 a=0,选择当前的访问存储区。如果 a=1,用 BSR 选择 GPR 存储区(默认)。

如果 a=0 且扩展指令集可用,只要 f≤95(5Fh),这条指令就会按立即数偏移量寻址模式执行。要获取更多的技术细节,请参阅 H.2.3 节。

字长： 1

周期： 1

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器 f	处理数据	写目的地址

例 1： SUBFWB REG, 1, 0

执行指令前，

REG = 3

W = 2

C = 1

执行指令后，

REG = FF

W = 2

C = 0

Z = 0

N = 1;结果为负数

例 2： SUBFWB REG,0,0

执行指令前，

REG = 2

W = 5

C = 1

执行指令后，

REG = 2

W = 3

C = 1

Z = 0

N = 0;结果为正数

例 3： SUBFWB REG, 1, 0

执行指令前，

REG = 1

W = 2

C = 0

执行指令后，

REG = 0

W = 2

C = 1

Z = 1

N = 0;结果为零

SUBLW 立即数减去 W 寄存器的内容

格式： SUBLW k

操作数： 0≤k≤255

操作： k-(W) → W

受影响的标志位： N,OV,C,DC,Z

编码：

0000	1000	kkkk	kkkk

描述： 将 8 位的立即数减去 W 寄存器的内容，结果放入 W 寄存器。

字长： 1

周期： 1

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读立即数 k	处理数据	写 W 寄存器

例 1： SUBLW 02h

执行指令前，
- W = 01h，
- C = ?

执行指令后，
- W = 01h
- C = 1；结果为正数
- Z = 0
- N = 0

例 2： SUBLW 02h

执行指令前，
- W = 02h，
- C = ?

执行指令后，
- W = 00h
- C = 1；结果为零
- Z = 1
- N = 0

例 3： SUBLW 02h

执行指令前，
- W = 03h，
- C = ?

执行指令后，
- W = FFh（二进制补码）
- C = 0；结果为负数
- Z = 0
- N = 1

SUBWF 将 f 寄存器的内容减去 W 寄存器的内容

格式： SUBWF f{,d{,a}}

操作数： 0≤f≤255
d∈[0,1]
a∈[0,1]

操作： (f)-(W)→ 目的地址

受影响的标志位： N,OV,C,DC,Z

编码：

0101	11da	ffff	ffff

描述： 从 f 寄存器中减去 W（二进制补码方式）。如果 d=0，结果放入 W。如果 d=1，结果放入 f 寄存器（默认）。
如果 a=0，选择当前的访问存储区。如果 a=1，用 BSR 选择 GPR 存储区（默认）。
如果 a=0 且扩展指令集可用，只要 f≤95（5Fh），这条指令就会按立即数偏移量寻址模式执行。要获取更多的技术细节，请参阅 H.2.3 节。

字长： 1

周期： 1

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器 f	处理数据	写目的地址

例 1： SUBWF REG, 1, 0

执行指令前，
- REG = 3
- W = 2
- C = ?

执行指令后，
- REG = 1
- W = 2
- C = 1；结果为正数
- Z = 0
- N = 0

例 2： SUBWF REG, 0, 0

执行指令前，
- REG = 2
- W = 2
- C = ?

执行指令后，
- REG = 2
- W = 0
- C = 1；结果为零
- Z = 1
- N = 0

例 3： SUBWF REG, 1, 0

执行指令前，
- REG = 1
- W = 2
- C = ?

执行指令后，
- REG = FFh；（二进制补码）
- W = 2
- C = 0；结果为负数
- Z = 0
- N = 1

SUBWFB 带借位将 f 寄存器的内容减去 W 寄存器的内容

格式： SUBWFB f{,d{,a}}

操作数： 0≤f≤255
d∈[0,1]
a∈[0,1]

操作： (f)−(W) − ($\overline{C}$)→ 目的地址

受影响的标志位： N,OV,C,DC,Z

编码：

0101	10da	ffff	ffff

描述： 从 f 寄存器中减去 W 寄存器的内容和借位(二进制补码形式)。如果 d=0,结果放入 W。如果 d=1,结果放入 f 寄存器(默认)。
如果 a=0,选择当前的访问存储区。如果 a=1,用 BSR 选择 GPR 存储区(默认)。
如果 a=0 且扩展指令集可用,只要 f≤95(5Fh),这条指令就会按立即数偏移量寻址模式执行。要获取更多的技术细节,请参阅 H.2.3 节。

字长： 1

周期： 1

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器 f	处理数据	写目的地址

例 1： SUBWFB REG, 1, 0

执行指令前，
REG = 19h (0001 1001)
W = 0Dh (0000 1101)
C = 1

执行指令后，
REG = 0Ch (0000 1011)
W = 0Dh (0000 1101)
C = 1
Z = 0
N = 0;结果为正数

SUBWFB 带借位将 f 寄存器的内容减去 W 寄存器的内容(续)

例 2： SUBWFB REG, 0, 0

执行指令前，
REG = 1Bh (0001 1011)
W = 1Ah (0001 1010)
C = 0

执行指令后，
REG = 1Bh (0001 1011)
W = 00h
C = 1
Z = 1;结果为零
N = 0

例 3： SUBWFB REG, 1, 0

执行指令前，
REG = 03h (0000 0011)
W = 0Eh (0000 1101)
C = 1

执行指令后，
REG = F5h (1111 0100)
(二进制补码)
W = 0Eh (0000 1101)
C = 0
Z = 0
N = 1;结果为负数

SWAPF　将 f 寄存器的内容做半字节交换

格式：　SWAPF f{,d{,a}}

操作数：　0≤f≤255
d∈[0,1]
a∈[0,1]

操作：　(f<3：0>)→目的地址<7：4>
(f<7：4>)→目的地址<3：0>

受影响的标志位：　无。

编码：

0011	10da	ffff	ffff

描述：　将 f 寄存器内容的高半字节和低半字节交换。如果 d=0，结果放入 W。如果 d=1，结果放入 f 寄存器(默认)。

如果 a=0，选择当前的访问存储区。如果 a=1，用 BSR 选择 GPR 存储区(默认)。

如果 a=0 且扩展指令集可用，只要 f≤95(5Fh)，这条指令就会按立即数偏移量寻址模式执行。要获取更多的技术细节，请参阅 H.2.3 节。

字长：　1

周期：　1

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器 f	处理数据	写目的地址

例：　SWAPF REG, 1, 0

执行指令前，
REG　=　53h

执行指令后，
REG　=　35h

TBLRD　读表

格式：　TBLRD (*; *+; *-; +*)

操作数：　无

操作：　如果执行 TBLRD*，那么
(Prog Mem(TBLPTR))→TABLAT
TBLPTR 的内容没有变化。
如果执行 TBLRD*+，那么
(Prog Mem(TBLPTR))→TABLAT
(TBLPTR)+1→TBLPTR
如果执行 TBLRD*-，那么
(Prog Mem(TBLPTR))→TABLAT
(TBLPTR)-1→TBLPTR
如果执行 TBLRD+*，那么
(TBLPTR)+1→TBLPTR
(Prog Mem(TBLPTR))→TABLAT

受影响的标志位：　无。

编码：

0000	0000	0000	10nn nn = 0 * = 1 * + = 2 * - = 3 + *

描述：　这条指令用来读取程序存储器(P.M.)中的内容。用来访问程序存储器的指针叫作表指针(TBLPTR)。

TBLPTT(21 位指针)指向程序存储器的每个字节。TBLPTR 有 2MB 地址范围。

TBLPTR[0]=0：程序存储器的低字节。

TBLPTR[0]=1：程序存储器的高字节。

TBLRD 可以改变 TBLPTR 的值的方式有：

- ☐ 不改变
- ☐ 后增量
- ☐ 后减量
- ☐ 前增量

TBLRD 读表(续)

字长： 1
周期： 2
Q 周期活动：

Q1	Q2	Q3	Q4
译码	无操作	无操作	无操作
无操作	无操作（读程序存储器）	无操作	无操作（写 TABLAT 寄存器）

例 1：TBLRD * + ；

执行指令前，

TABLAT	=	55h
TBLPTR	=	00A356h
存储器(00A356h)	=	34h

执行指令后，

TABLAT	=	34h
TBLPTR	=	00A357h

例 2：TBLRD + * ；

执行指令前，

TABLAT	=	0AAh
TBLPTR	=	01A357h
存储器(01A357h)	=	12h
存储器(01A358h)	=	34h

执行指令后，

TABLAT	=	34h
TBLPTR	=	01A358h

TBLWT 写表

格式： TBLWT (* ; * + ; * − ; + *)

操作数： 无

操作： 如果执行 TBLWT * ，那么：
TABLAT→保持寄存器
TBLPTR 的内容没有变化
如果执行 TBLWT * +，那么：
TABLAT→保持寄存器
(TBLPTR)+1 →TBLPTR
如果执行 TBLWT * −，那么：
TABLAT→保持寄存器
(TBLPTR)−1 →TBLPTR
如果执行 TBLWT + * ，那么：
(TBLPTR)+1 →TBLPTR
TABLAT→保持寄存器

受影响的标志位： 无。

编码：

0000	0000	0000	11nn nn = 0 * = 1 * + = 2 * − = 3 + *

描述： 这条指令用 TBLPTR 的 3 个最低位来确定写入 TABLAT 的 8 个保留寄存器的哪一个。保持寄存器用来对程序存储器的内容进行编程。(请参阅第 6 章。)
TBLPTR(21 位指针)指向程序存储器的每个字节。TBLPTR 有 2MB 地址范围。
TBLPTR[0]=0：程序存储器的低字节。
TBLPTR[0]=1：程序存储器的高字节。
TBLRD 可以改变 TBLPTR 的值的方式有：
- ☐ 不改变
- ☐ 后增量
- ☐ 后减量
- ☐ 前增量

TBLWT 写表(续)

字长： 1

周期： 2

Q 周期活动：

Q1	Q2	Q3	Q4
译码	无操作	无操作	无操作
无操作	无操作(读 TABLAT)	无操作	无操作(写保持寄存器)

例 1： `TBLWT * + ;`

执行指令前，

TABLAT	=	55h
TBLPTR	=	00A356h
保持存储器(00A356h)	=	FFh

执行指令后，

TABLAT	=	55h
TBLPTR	=	00A357h
保持存储器(00A356h)	=	55h

例 2： `TBLWT + * ;`

执行指令前，

TABLAT	=	34h
TBLPTR	=	01389Ah
保持存储器(01389Ah)	=	FFh
保持存储器(01389Bh)	=	FFh

执行指令后，

TABLAT	=	34h
TBLPTR	=	01389Bh
保持存储器(01389Ah)	=	FFh
保持存储器(01389Bh)	=	34h

TSTFSZ 测试 f，若为 0 则跳过

格式： TSTFSZ f{,a}

操作数： 0≤f≤255

a∈[0,1]

操作： 若 f=0，则跳过

受影响的标志位： 无。

编码：

0110	011a	ffff	ffff

描述： 如果 f 寄存器的内容为 0，放弃当前执行的指令，取而代之的是执行一条 NOP 指令，使其成为 2 周期指令。

如果 a=0，选择当前的访问存储区。如果 a=1，用 BSR 选择 GPR 存储区(默认)。

如果 a=0 且扩展指令集可用，只要 f≤95(5Fh)，这条指令就会按立即数偏移量寻址模式执行。要获取更多的技术细节，请参阅 H. 2. 3 节。

字长： 1

周期： 1(2)

备注：如果跳过且紧跟的是双字指令，需要占用 3 个指令周期。

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器 f	处理数据	无操作

如果跳过：

Q1	Q2	Q3	Q4
无操作	无操作	无操作	无操作

如果跳过，并紧接一条双字指令：

Q1	Q2	Q3	Q4
无操作	无操作	无操作	无操作
无操作	无操作	无操作	无操作

例：

```
HERE TSTFSZ CNT, 1
NZERO:
ZERO:
```

执行指令前，

PC	=	地址(HERE)

执行指令后，

如果 CNT	=	00h
PC	=	地址(ZERO)

如果 CNT 不等于 00h

PC	=	地址(NZERO)

XORLW 立即数与W异或

格式： XORLW k

操作数： 0≤k≤255

操作： (W)异或 k → W

受影响的标志位： N,Z

编码：

0000	1010	kkkk	kkkk

描述： 将W寄存器的内容和8位的立即数做逻辑异或运算，结果放入W寄存器。

字长： 1

周期： 1

Q周期活动：

Q1	Q2	Q3	Q4
译码	读立即数 k	处理数据	写入W寄存器

例： XORLW 0AFh

执行指令前，
W = B5h

执行指令后，
W = 1Ah

XORWF W和f异或

格式： XORWF f{,d{,a}}

操作数： 0≤f≤255
d∈[0,1]
a∈[0,1]

操作： (W)异或(f) → 目的地址

受影响的标志位： N,Z

编码：

0001	10da	ffff	ffff

描述： 将W寄存器的内容和f寄存器的内容进行异或运算。如果d=0，结果放入W。如果d=1，结果放入f寄存器(默认)。
如果a=0，选择当前的访问存储区。
如果a=1，用BSR选择GPR存储区(默认)。
如果a=0且扩展指令集可用，只要f≤95(5Fh)，这条指令就会按立即数偏移量寻址模式执行。要获取更多的技术细节，请参阅H.2.3节。

字长： 1

周期： 1

Q周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器 f	处理数据	写目的地址

例： XORWF REG, 1, 0

执行指令前，
REG = AFh
W = B5h

执行指令后，
REG = 1Ah
W = B5h

797 ~ 798

H.2 扩展指令集

在 PIC18 的标准 75 条指令的集合基础上，PIC18F2480/2580/4480/4580 还提供了可选的核心 CPU 功能的扩展功能。这些增加的功能包括 8 条新增的指令，它们支持间接和变址寻址方式，以及用于许多标准的 PIC18 指令的变址立即数偏移寻址模式。

这些新增的功能在默认情况下是禁用的。若要启用它们，则需要把 XINST 配置位置 1。

扩展指令集里的指令都可以看成立即数操作，它们可用于访问文件选择寄存器或者用作变址寻址。其中的两条指令（ADDFSR 和 SUBFSR）都是新增用于操作 FSR2 的特殊指令。一些指令（如 ADDULNK 和 SUBULNK）还允许在执行指令后自动返回。

扩展指令集是对高级语言程序（特别是 C 语言）的再植入程序代码（一种递归代码或者是用于软件栈）的优化补充。另外，它们允许用户使用高级语言更高效地操作数据结构，包括：

- 在进入和离开子例程时，软件栈空间的动态地址分配和回收；
- 函数指针调用；
- 软件栈指针操作；
- 对软件栈中变量的调用。

表 H-3 对扩展指令集做了小结。更多的描述将在 H.2.2 节中介绍。表 H-1 中的操作码字符对 PIC18 的标准指令集或扩展指令集都适用。

> **注意：**扩展指令集和变址立即数偏移寻址模式都是为优化 C 语言程序而引入的。用户可能不会在编译器里直接使用这些指令。这些命令的格式仅供用户在检查编译器产生的代码时作为参考。

H.2.1 扩展指令格式

大部分的扩展指令都使用变址变量（也就是一个文件选择寄存器和其他偏移量）来指定源寄存器或目标寄存器。当一条指令的变量用作变址寻址的一部分时，它会被标上方括号（[ ]）。这样做的目的是为了表示变量用作变址或者偏移量。如果变址或者偏移量没有被标上方括号，那么 MPASM™ 编译器将其标记为一个错误。

当扩展指令集可用时，方括号也通常用来表示面向字节的指令和面向位的指令的变址变量。这是指令格式里的一点点变动。要了解更详细的技术细节，请参阅 H.2.3 节。

> **注意：**在以前的 PIC18 和早期指令系统中，方括号用来表示可选变量。在本章和以后的资料中，可选变量均用中括号（{ }）来表示。

表 H-3 PIC18 指令集的扩展

助记符，操作数	描 述	周 期	16 位指令字				受影响的标志位
			MSB			LSB	
ADDFSR f,k	将立即数与 FSR 相加	1	1110	1000	ffkk	kkkk	无
ADDULNK k	将立即数加到 FSR2 中并返回	2	1110	1000	11kk	kkkk	无
CALLW	使用 WREG 调用子例程	2	0000	0000	0001	0100	无
MOVSF z_s,f_d	将 z_s(源地址)(第一个字)传送到 f_d(目的地址)(第二个字)	2	1110 1111	1011 ffff	0zzz ffff	zzzz ffff	无
MOVSS z_s,f_d	将 z_s(源地址)(第一个字)传送到 f_d(目的地址)(第二个字)	2	1110 1111	1011 xxxx	1zzz xzzz	zzzz zzzz	无
PUSHL k	保存立即数到 FSR,FSR 自减 1	1	1110	1010	kkkk	kkkk	无
SUBFSR f,k	从 FSR 中减去立即数	1	1110	1001	ffkk	kkkk	无
SUBULNK k	从 FSR 中减去立即数并返回	2	1110	1001	11kk	kkkk	无

H.2.2 扩展指令集

ADDFSR 将立即数加到 FSR 中

格式： ADDFSR f,k

操作数： 0≤f≤63
f∈[0,1,2]

操作： FSR(f)+k → FSR(f)

受影响的标志位： 无。

编码：

1110	1000	ffkk	kkkk

描述： 将 6 位的立即数加到由 f 指定的 FSR 中。

字长： 1

周期： 1

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读立即数 k	处理数据	写入 FSR 寄存器

例： ADDFSR 2, 23h

执行指令前，
FSR2 = 03FFh

执行指令后，
FSR2 = 0422h

ADDULNK 将立即数加到 FSR2 中，然后返回

格式： ADDULNK k

操作数： 0≤f≤63

操作： FSR2+2 → FSR2
PC=(TOS)

受影响的标志位： 无。

编码：

1110	1000	11kk	kkkk

描述： 将 6 位的立即数加到 FSR2 中，然后执行 RETURN，将 TOS 装载到 PC。该指令占用 2 个指令周期。在第二个指令周期执行一条 NOP 指令。
这可以看作是 ADDFSR 指令在 f=3（二进制 11）时的特殊情况；它只对 FSR2 进行操作。

字长： 1

周期： 2

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读立即数 k	处理数据	写 FSR 寄存器
无操作	无操作	无操作	无操作

例： ADDULNK 23h

执行指令前，
FSR2 = 03FFh
PC = 0100h
TOS = 02AFh

执行指令后，
FSR2 = 0422h
PC = 02AFh
TOS = TOS−1

注意：所有 PIC18 指令在指令伪代码前都可使用可选的标号变量作为符号寻址。当使用标号时，指令格式将变成：{标号} 指令变量。

CALLW 使用 WREG 调用子例程

格式： CALLW

操作数： 无

操作： (PC+2) → TOS
(W) → PCL
(PCLATH) → PCH
(PCLATU) → PCU

受影响的标志位： 无。

编码：

0000	0000	0001	0100

描述： 首先，将返回地址(PC+2)压栈。接下来，将 W 寄存器的内容写入 PCL，放弃原有的值。然后，将 PCLATH 和 PCLATU 的内容锁入 PCH 和 PCU。在第二个指令周期执行一条 NOP 指令，而获取新的指令。
与 CALL 指令不同，这里不能选择 W、状态位和 BSR。

字长： 1

周期： 2

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读 WREG	将 PC 压栈	无操作
无操作	无操作	无操作	无操作

例： HERE CALLW

执行指令前，
PC = 地址(HERE)
PCLATH = 10h
PCLATU = 00h
W = 06h

执行指令后，
PC = 001006h
TOS = 地址(HERE+ 2)
PCLATH = 10h
PCLATU = 00h
W = 06h

MOVSF 将变址传送给 f

格式： MOVSF [z_s], f_d

操作数： $0 \leqslant z_s \leqslant 127$
$0 \leqslant f_d \leqslant 4095$

操作： ((FSR2)+z_s) → f_d

受影响的标志位： 无。

编码：

1110	1011	0zzz	zzzz$_s$
1111	ffff	ffff	ffff$_d$

描述： 将源寄存器的内容传送到目标寄存器 f_d 中。源寄存器的实际地址是 FSR2 的第一字加上 7 位立即数偏移量。目的寄存器地址是第二字的 12 位立即数 f_d。所有地址都可以是 4096B 数据空间范围(000h～FFFh)内的任何位置。
MOVSF 指令不能使用 PCL、TOSU、TOSH 或者 TOSL 作为目的寄存器。
如果源地址指向直接寻址寄存器，返回值将是 00h。

字长： 2

周期： 2

Q 周期活动：

Q1	Q2	Q3	Q4
译码	确定源地址	确定源地址	读源寄存器
译码	无操作，无哑读	无操作	写 f 寄存器（目的地址）

例： MOVSF [05h], REG2

执行指令前，
FSR2 = 80h
85h 的内容 = 33h
REG2 = 11h

执行指令后，
FSR2 = 80h
85h 的内容 = 33h
REG2 = 33h

MOVSS 从变址寄存器传送到变址寄存器

格式： MOVSS [z_s],[z_d]

操作数： $0 \leqslant z_s \leqslant 127$

$0 \leqslant z_d \leqslant 127$

操作： $((FSR2)+z_s) \rightarrow ((FSR2)+z_d)$

受影响的标志位： 无。

编码：

第一个字(源地址)	1110	1011	1zzz	$zzzz_s$
第二个字(目的地址)	1111	xxxx	xzzz	$zzzz_d$

描述： 将源寄存器的内容传送到目的寄存器。源寄存器和目的寄存器的实际地址分别由 FSR2 的第一字加上 7 位立即数偏移量 z_s 或者 z_d 来确定。所有地址都可以是 4096B 数据空间范围(000h～FFFh)内的任何位置。

MOVSF 指令不能使用 PCL、TOSU、TOSH 或者 TOSL 作为目的寄存器。

如果源地址指向直接寻址的寄存器，返回值为 00h。如果目的地址指向直接寻址的寄存器，将执行一条 NOP 指令。

字长： 2

周期： 2

Q 周期活动：

Q1	Q2	Q3	Q4
译码	确定源地址	确定源地址	读源寄存器
译码	确定目的地址	确定目的地址	写目的寄存器

例： MOVSS [05h],[06]

执行指令前，

FSR2 = 80h

85h 的内容 = 33h

86h 的内容 = 11h

执行指令后，

FSR2 = 80h

85h 的内容 = 33h

86h 的内容 = 33h

PUSHL 保存立即数到 FSR2，FSR 减 1

格式： PUSHL k

操作数： $0 \leqslant k \leqslant 255$

操作： k → (FSR2)

FSR2−1 → FSR2

受影响的标志位： 无。

编码：

1111	1010	kkkk	kkkk

描述： 将 8 位的立即数写入到 FSR2 指定的数据存储器地址，然后将 FSR2 减 1。

这条指令允许用户把数值压入软件栈。

字长： 1

周期： 1

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读立即数 k	处理数据	写目的寄存器

例： PUSHL 08h

执行指令前，

FSR2H：FSR2L = 01ECh

存储器(01ECh) = 00h

执行指令后，

FSR2H：FSR2L = 01EBh

存储器(01ECh) = 08h

SUBFSR 从 FSR 中减去立即数

格式： SUBFSR f,k

操作数： 0≤k≤63
f∈[0,1,2]

操作： FSRf−k → FSRf

受影响的标志位： 无。

编码：

1110	1001	ffkk	kkkk

描述： 将由 f 指定的 FSR 的内容减去 6 位的立即数。

字长： 1

周期： 1

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器 f	处理数据	写目的地址

例： SUBFSR 2, 23h

执行指令前，
FSR2 = 03FFh

执行指令后，
FSR2 = 03DCh

SUBULNK 从 FSR2 中减去立即数并返回

格式： SUBULNK k

操作数： 0≤k≤63

操作： FSR2−k → FSR2
(TOS)→ PC

受影响的标志位： 无。

编码：

1110	1001	11kk	kkkk

描述： 将 FSR2 的内容减去 6 位的立即数。然后将 TOC 的内容加载给 PC，并执行 RETURN 操作。
该指令占用 2 个指令周期。在第二个周期执行一条 NOP 指令。
该指令可以看作 SUBFSR 指令在 f=3（二进制 11）时的特殊情况；它只对 FSR2 进行操作。

字长： 1

周期： 2

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器 f	处理数据	写目的地址
无操作	无操作	无操作	无操作

例： SUBULNK 23h

执行指令前，
FSR2 = 03FFh
PC = 0100h

执行指令后，
FSR2 = 03DCh
PC = (TOS)

H.2.3 变址立即数偏移寻址模式中的面向字节和面向位的指令

除了扩展集里的8条新命令，启用扩展指令系统的同时，也启用了变址立即数偏移量寻址模式(如5.6.1节所述)。这将对PIC18的很多标准指令的解释方式产生影响。

当扩展指令系统被禁止时，操作码所包含的地址将被当作立即数存储地址：要么是当前的访问存储区地址(a=0)，要么是由BSR指定的GPR存储区(a=1)。如果扩展指令集被启用且a=0，那么文件寄存器变量在不大于5Fh时会被解释为FSR2的指针偏移量，而不是立即数的地址。在实际中，使用访问存储区RAM位作为变量的所有指令，也就是所有面向字节和面向位的指令，即总共有超过半数的核心PIC18指令都会因为扩展指令集的启用而改变工作方式。当FSR2的内容为00h时，访问RAM的边界被重新确定为它们的初值。这在生成反向的兼容性代码时是非常有用的。如果使用这种方法，那么就很有必要保存FSR2的值，并在C和汇编程序间转换时恢复它，以保护栈指针。用户必须记住扩展指令集的指令格式的要求。

虽然变址立即数偏移寻址模式对于动态栈和指针的操作非常有用，但是对一个错误的寄存器执行简单的算术操作将会惹来麻烦。熟悉PIC18编程的用户必须记住，当启用了扩展指令集时，不大于5fh的寄存器地址将被用作变址立即数偏移量寻址。

在下文中给出了几个典型的例子，来说明面向字节的和面向位的指令在变址立即数偏移寻址模式下执行时会受到怎样的影响。例子中给出的操作数条件适用于所有这类指令。

PIC18标准指令的扩展指令格式

当扩展指令集被启用时，标准的面向字节和面向位的命令中的文件寄存器变量f，将被立即数偏移量k代替。正如前面介绍的，这种情况只有在f小于或等于5Fh时才会发生。当使用偏移量时，必须使用方括号([])作为标记。在执行扩展指令时，编译器将把方括号中的数值解释为变址或偏移量。如果没有使用括号，或者括号里的数值大于5Fh，那么在MPASM™编译器中将会出现错误。

如果变址立即数偏移寻址的变址变量被正确标识，那么访问RAM变量将从来不会被指定；它将自动地被设为0。这恰恰同标准指令(扩展指令集禁用时)中对目标地址的a置1的情况相反。在这个模式下，若指定了访问寄存器位，也将在MPASM™编译器里产生错误。

目标变量d的作用和以前介绍的一样。最新的MPASM编译器版本里，扩展指令的支持语言要明确声明。这可以在源文件中使用命令行的选项“/y”或者PE伪指令实现。

H.2.4 启用扩展指令集的注意事项

要特别注意的是，扩展指令不是对所有用户都有利的。特别是那些不对软件栈编写程序的用户，可能就不会从扩展指令集中得到便利。

另外，变址立即数偏移寻址模式可能会对PIC18编译器的合法程序产生影响。这是因为合法代码中的指令可能会访问5Fh以下的地址。由于在启用扩展指令集时，这些地址被解释为FSR2的立即数偏移量，所以在应用中会导致数据地址的读写错误。

当向PIC18F2480/2580/4480/4580接入一个应用时，代码类型很重要。一个用C写的、可再植入的大程序会在使用扩展指令集时能从高效编译环境中受益。主要使用访问存储区的合法应用好像最不可能从扩展指令集中受益。

ADDWF 将 W 的内容加到变址寄存器（变址立即数偏移寻址模式）

格式： ADDWF [k],{,d}

操作数： 0≤k≤95
d∈[0,1]
a=0

操作： (W)+((FSR2)+k) → 目的地址

受影响的标志位： N,OV,C,DC,Z

编码：

0010	01d0	kkkk	kkkk

描述： 将 W 寄存器的内容加到由 FSR2(偏移量为 k)指定的寄存器中。
如果 d=0，结果放入 W。如果 d=1，结果放入 f 寄存器(默认)。

字长： 1

周期： 1

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读立即数 k	处理数据	写目的地址

例： ADDWF [OFST], 0

执行指令前，

W = 17h
OFST = 2Ch
FSR2 = 0A00h
0A2Ch 的内容 = 20h

执行指令后，

W = 37h
0A2Ch 的内容 = 20h

BSF 对变址寄存器的位置 1（变址立即数偏移寻址模式）

格式： BSF [k],b

操作数： 0≤k≤95
0≤b≤7
a=0

操作： 1 → ((FSR2)+k)<b>

受影响的标志位： 无。

编码：

1000	bbb0	kkkk	kkkk

描述： 将由 FSR2(偏移量为 k)指定的寄存器位 b 置 1。

字长： 1

周期： 1

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读寄存器 f	处理数据	写目的地址

例： BSF [FLAG_OFST], 7

执行指令前，

FLAG_OFST = 0Ah
FSR2 = 0A00h
0A0Ah 的内容 = 55h

执行指令后，

0A0Ah 的内容 = D5h

注意：使用 PIC18 扩展指令集可能会导致合法的操作执行不完整或者执行完全失败。

SETF 对变址寄存器的内容置 1 (变址立即数偏移寻址模式)

格式： SETF [k]

操作数： 0≤k≤95

操作： FFh → ((FSR2)+k)

受影响的标志位： 无。

编码：

0110	1000	kkkk	kkkk

描述： 将由 FSR2(偏移量为 k)指定的寄存器的内容置为 FFH。

字长： 1

周期： 1

Q 周期活动：

Q1	Q2	Q3	Q4
译码	读立即数 k	处理数据	写寄存器

例： SETF [OFST]

执行指令前，

OFST = 2Ch

FSR2 = 0A00h

0A2Ch 的内容 = 00h

执行指令后，

0A2Ch 的内容 = FFh

H.2.5 使用 Microchip MPLAB® IDE 工具的注意事项

Microchip 公司的最新软件工具是为支持全部的 PIC18F2480/2580/4480/4580 系列的扩展指令而设计的，这包括了 MPLAB C18 C 编译器，MPASM 汇编语言和 MPLAB 集成开发环境(IDE)。

在选择合适的器件以开发软件时，MPLAB IDE 会自动把器件的配置位设为默认值。XINST 的配置位的默认值为 0，即禁用扩展指令系统和变址立即数偏移寻址模式。为了使用扩展指令集来设计合理的程序，在编程过程中必须将 XINST 置 1。

要使用扩展指令集开发软件，用户必须在语言工具里启用指令和变址立即数偏移寻址模式的支持项。根据不同的环境，有以下几种设置方法。

- ❑使用开发环境下的菜单项或者对话框来配置软件开发的语言工具和项目的设置。
- ❑使用命令行选项。
- ❑在源代码中使用伪指令。

这些选项在不同的编译器和开发环境中会有很大的不同。用户可以查看开发环境自带的文件来获得适用的信息。

806



索 引

索引中的页码为英文原书页码，与本书页边标注的页码一致。

A

B

C

D

K

L

M

N

O

P

R

S

T

U

W~Z